

# Automates d'arbres et réécriture pour l'étude de problèmes d'accessibilité

## THÈSE

présentée et soutenue publiquement le 19 décembre 2003

pour l'obtention du

Doctorat de l'Université de Rennes I  
spécialité informatique

par

Valérie Viêt Triêm Tông

### Composition du jury

*Rapporteurs* : Jean GOUBAULT-LARRECQ  
Pierre RÉTY

*Examineurs* : Jean Pierre BANÂTRE  
Thomas GENET  
Thomas JENSEN  
Francis KLAY

Mis en page avec la classe thloria.

*à mon père*



# Table des matières

## Chapitre 1

### Introduction

1.1	Motivations . . . . .	6
1.2	Cadre de travail . . . . .	6
1.3	Plan de la thèse . . . . .	7

## Partie I Automates et Réécriture

9

## Chapitre 2

### Preliminaires

2.1	Termes et réécriture . . . . .	12
2.1.1	Rappel sur les relations d'ordre . . . . .	14
2.2	Automates d'arbres . . . . .	15
2.2.1	Automates d'arbres . . . . .	15
2.2.2	Opérations usuelles . . . . .	16

## Chapitre 3

### Complétion d'automates

3.1	Compléter $\Delta$ par $\mathcal{R}$ . . . . .	20
3.1.1	Normalisations de transitions . . . . .	22
3.2	Automate clos pour un système de réécriture . . . . .	24
3.2.1	Condition de linéarité . . . . .	24
3.2.2	Condition pour que $\mathcal{A}_{\mathcal{R},\alpha}^*$ existe toujours . . . . .	27
3.3	Filtrage dans un automate . . . . .	28
3.3.1	Algorithmes existants . . . . .	28
3.3.2	Algorithme proposé . . . . .	29
3.3.3	Comparaison des algorithmes, complexité du problème . . . . .	34
3.4	Rôle de l'approximation . . . . .	34

3.4.1	Choix de la fonction d'abstraction . . . . .	35
3.4.2	Calcul exact des descendants . . . . .	37
3.5	Utiliser la complétion d'automates pour des preuves de propriétés . . . . .	39

**Partie II Validation de protocoles cryptographiques par réécriture 43**

**Chapitre 4**  
**Introduction à la cryptographie**

4.1	Un peu de cryptographie . . . . .	46
4.1.1	la cryptographie à clé secrète . . . . .	46
4.1.2	la cryptographie à clé publique . . . . .	48
4.2	Utilisation de la cryptographie . . . . .	51
4.2.1	Buts actuels . . . . .	51
4.2.2	Quelles sont les menaces? . . . . .	51
4.3	Vérification de protocoles cryptographiques . . . . .	51
4.3.1	Modélisation classique . . . . .	51
4.4	Modélisation proposée : généralités . . . . .	53

**Chapitre 5**  
**Application à la vérification de protocoles**

5.1	Vérification du protocole <i>View Only</i> de <b>SmartRight</b> . . . . .	56
5.1.1	Présentation du protocole . . . . .	56
5.1.2	Spécification du protocole . . . . .	59
5.1.3	Les configurations initiales . . . . .	61
5.2	SmartRight et l'intrus . . . . .	63
5.2.1	Rejeu . . . . .	63
5.2.2	Un intrus capable de chiffrer, déchiffrer, composer et décomposer . . .	65
5.2.3	Un intrus connaissant $\mathcal{K}$ . . . . .	66
5.2.4	Un intrus générant des <i>Control Word</i> . . . . .	67
5.3	Conclusion . . . . .	67

**Partie III Conjectures négatives 69**

**Chapitre 6**  
**Théories, modèles et preuves**

6.1	Théories & modèles . . . . .	72
-----	------------------------------	----

6.1.1	Langages, termes et formules . . . . .	72
6.1.2	Interprétation et satisfaction . . . . .	73
6.1.3	Modèles de Herbrand . . . . .	75
6.1.4	Théorie inductive . . . . .	76
6.2	Méthodes de preuves . . . . .	78
6.2.1	Récurrance explicite . . . . .	78
6.2.2	Preuve par cohérence . . . . .	78
6.2.3	Récurrance par réécriture . . . . .	82

## Chapitre 7

### Preuves de propriétés initiales

7.1	Principe de l'approche . . . . .	86
7.1.1	Exemple introductif . . . . .	86
7.1.2	Système de preuve . . . . .	88
7.1.3	Correction de l'algorithme . . . . .	89
7.1.4	Un exemple détaillé . . . . .	91
7.2	Preuves de propriétés initiales par complétion d'automate . . . . .	93
7.2.1	Modélisation du système . . . . .	93
7.2.2	Principe général . . . . .	95
7.2.3	Reprenons notre exemple . . . . .	95
7.2.4	Limite de l'approche . . . . .	96
7.3	Vers l'interprétation abstraite . . . . .	96
7.4	Et la complétude réfutationnelle? . . . . .	97

## Chapitre 8

### Bilan et perspectives

8.1	Bilan . . . . .	100
8.1.1	Extension de la complétion aux systèmes non linéaires et calcul exact . . . . .	100
8.1.2	Application à la vérification de protocoles cryptographiques . . . . .	100
8.1.3	Un système de preuve de propriétés initiales . . . . .	100
8.2	Perspectives . . . . .	101

## Annexes

**103**

### Annexe A Implémentation de l'algorithme de filtrage

**103**

A.1	Construction de l'automate d'un terme . . . . .	104
A.2	calcul des transitions utiles . . . . .	106
A.3	calcul des $Q$ -substitutions . . . . .	108

<b>Annexe B Fonctions d'abstraction</b>	<b>113</b>
B.1 Fonction d'abstraction d'un polynôme . . . . .	114
B.2 Règles d'abstraction pour <i>View Only</i> sans intrus . . . . .	116
<b>Annexe C Automate complet pour SmartRight sans intrus</b>	<b>119</b>
C.1 Automate complet pour <i>View Only</i> . . . . .	120
<b>Annexe D Automate initial pour <i>deriv</i></b>	<b>127</b>
D.1 Automate initial . . . . .	128
<b>Table des figures</b>	<b>129</b>
<b>Index</b>	<b>131</b>
<b>Bibliographie</b>	<b>137</b>

1

# Introduction

## 1.1 Motivations

Le 20 juin 1999, une interview de Serge Humpich démontrant par l'exemple que la carte bleue était falsifiable était programmée au "Vrai Journal" de Karl Zero. Serge Humpich souhaitait montrer qu'il était possible de créer de nouvelles cartes bancaires, ne correspondant à aucun compte bancaire réel mais reconnues par des terminaux de paiement électronique chez les commerçants comme une carte émanant d'une autorité bancaire. Pour faire sa démonstration, S. Humpich a acheté à la RATP dix carnets de tickets de métro, la RATP a bien été créditée de l'opération mais les banques n'ont pu imputer l'opération à un compte bancaire de porteur réel. Le GIE Carte Bleue a fait censurer cette émission. Le 25 janvier 2000, le tribunal correctionnel de Paris a condamné S. Humpich à dix mois de prison avec sursis.

Cette affaire a mis en lumière des lacunes manifestes au niveau de la sécurité des paiements. Les répercussions financières et économiques sont très importantes : il semble inévitable de modifier tous les terminaux de paiement, voire de les remplacer. Le budget se chiffre en dizaines de millions d'euros.

Cette fissure dans la sécurité bancaire est autrement connue sous le nom de *YesCard*, il s'agit en réalité d'une carte bancaire factice permettant d'effectuer des achats de faible montant (moins de 100 euros). Aujourd'hui une petite recherche sur internet permet d'apprendre que pour fabriquer une *YesCard*, il suffit de posséder :

- un lecteur-encodeur de carte à puce,
- une carte à puce vierge autrement appelée *goldwaffer*,
- deux ou trois logiciel permettant de programmer la carte vierge.

Un lecteur-encodeur est accessible à partir de 45 euros environ, une carte vierge coûte 10 euros et les logiciels sont tous disponibles sur internet.

La faiblesse de la carte bancaire n'est pas d'ordre matériel<sup>1</sup>, mais d'ordre cryptographique. Un terminal pour carte bleue utilise des protocoles pour authentifier la carte, son porteur et le compte en banque auquel elle correspond. Le problème est que ces protocoles ne sont pas sûrs. La faiblesse de ces protocoles n'est pas, contrairement à ce qui vient immédiatement à l'esprit, une faiblesse des algorithmes de chiffrements<sup>2</sup>, en réalité beaucoup d'attaques reposent sur des manipulations de messages comme l'interception d'un message et le renvoi d'un autre. Le domaine d'étude qui nous intéresse ici est la sécurité des protocoles cryptographiques et plus particulièrement l'accessibilité des informations sensibles. Nous proposons une méthode de vérification dont le principe est de calculer un sur-ensemble des situations possibles, puis de rechercher si des situations interdites ont été produites ou non.

## 1.2 Cadre de travail

Nous décrivons les protocoles cryptographiques à l'aide de règles de réécriture agissant sur des termes représentant les configurations du protocole. Plus précisément, nous utilisons des langages réguliers et des règles de réécriture pour modéliser les protocoles. L'intérêt des langages réguliers est qu'ils sont représentables par des automates d'arbres, ce qui nous permet de représenter de manière finie un ensemble potentiellement infini de situations dans un protocole. Parallèlement,

---

<sup>1</sup>bien que sur internet on vous explique comment trouver les codes d'une carte en la trempant dans l'eau bouillante...

<sup>2</sup>d'ailleurs une hypothèse classique de vérification est le chiffrement parfait

nous décrirons toutes les actions possibles (règles du protocole et comportement de l'intrus) par un système de réécriture  $\mathcal{R}$ . Le principe de la vérification est de représenter les situations conflictuelles par des termes interdits, la sécurité du protocole est assurée lorsque ces termes sont inatteignables par  $\mathcal{R}$ . Cette approche s'appuie sur

- la représentation par automates d'arbres des situations du protocole,
- un algorithme de complétion à la *Knuth Bendix* pour le calcul des descendants,
- une méthode de sur-approximation dans les automates qui permet de *forcer la terminaison* du processus de complétion.

Nous avons utilisé cette méthode pour vérifier le protocole *View Only* de **SmartRight**, un protocole de protection de contenu numérique développé par *Thomson Multimédia*. Cette étude nous a permis de montrer que la représentation par automates était particulièrement bien adaptée pour vérifier des propriétés *atypiques* comme c'était le cas de la propriété d'*Anti Replay* de **SmartRight**.

Plus généralement, notre procédé permet de résoudre des problèmes d'atteignabilités dans les langages réguliers sur lesquelles s'appliquent des théories équationnelles orientables en systèmes de réécritures confluents. Autrement dit, si  $E_1$  et  $E_2$  sont des ensembles finis de termes clos, ou plus généralement des langages réguliers, notre méthode est une procédure de semi décision pour savoir si  $\mathcal{R}^*(E_1)$  et  $\mathcal{R}^*(E_2)$ <sup>3</sup> ont des termes communs ou non.

Nous nous sommes ensuite intéressé à savoir s'il était possible d'étendre ce raisonnement pour montrer de manière automatique des diséquations, c'est à dire des propriétés de la forme

$$\forall x_1 \dots x_n (t_1 \neq t_2)$$

C'est un problème à ne pas confondre avec les preuves de théorèmes inductifs. Les théorèmes inductifs sont les équations qui sont vraies dans l'ensemble des modèles de Herbrand d'une théorie équationnelle donnée; les diséquations ne sont en général vraies que dans le modèle initial ou plus petit modèle de Herbrand.

### 1.3 Plan de la thèse

Cette thèse est divisée en trois parties :

- La première partie étudie la complétion d'automates : nous rappellerons les notions de réécriture et d'automates d'arbres dont nous aurons besoin tout au long de ce document avant de présenter la complétion d'automates pour laquelle nous avons développé un algorithme améliorant les algorithmes existants.
- La seconde partie est une introduction à la cryptographie suivie de l'étude détaillée du protocole *View Only* de **SmartRight** : modélisation du protocole à l'aide d'automates et règles de réécriture, puis vérification des propriétés de secret, authentification et *Anti Replay*.
- La dernière partie de ce document s'intéresse à la preuve automatique de propriétés initiales. Nous étudions dans le chapitre 6 les méthodes de démonstrations automatiques : récurrence explicite, récurrence par réécriture et preuve par cohérence. Le dernier chapitre présente en premier lieu un algorithme de preuve de propriétés initiales, avant de

<sup>3</sup>ensembles des termes atteignables par  $\mathcal{R}$  à partir de  $E_1$  et  $E_2$

montrer comment il est possible d'utiliser la complétion d'automates pour automatiser cet algorithme.

Première partie

**Automates et Réécriture**



2

# Préliminaires

## 2.1 Termes et réécriture

Nous rappelons ici les notions essentielles sur la réécriture de termes et les automates d'arbres qui nous seront utiles pour la suite du document. Notre lecteur trouvera une documentation plus riche dans [DJ90, CDG<sup>+</sup>97]

**Définition 1 (Signature sortée)** Une signature est un couple  $\Sigma = (\mathcal{S}, \mathcal{F})$  où  $\mathcal{S}$  est un ensemble non vide symboles appelés sorties et  $\mathcal{F}$  un ensemble de symboles de fonctions distinct de  $\mathcal{S}$  associés à un profil :

$$f : \mathcal{S}_1 \times \dots \times \mathcal{S}_n \rightarrow \mathcal{S}_{n+1} \quad (n \geq 0, \mathcal{S}_i \in \mathcal{S} (0 \leq i \leq n)).$$

$\mathcal{S}_1 \times \dots \times \mathcal{S}_n$  est appelé domaine de  $f$ , et  $\mathcal{S}_{n+1}$  codomaine, et  $n$  arité de  $f$ . Une constante est un symbole de fonction d'arité 0.  $\mathcal{F}$  est aussi appelé alphabet.

Dans toute la première partie nous n'utiliserons pas les signatures sortées, plus simplement nous supposerons que  $\mathcal{S}$  est réduit à un élément. Un alphabet  $\mathcal{F}$  est alors simplement un ensemble fini de symboles associé à une fonction d'arité :  $ar : \mathcal{F} \rightarrow \mathbb{N}$ .

**Définition 2 (Termes)** Soient  $\mathcal{F}$  un alphabet et  $\mathcal{X}$  un ensemble de variables, l'ensemble des termes :  $\mathcal{T}(\mathcal{F}, \mathcal{X})$  est le plus petit ensemble tel que :

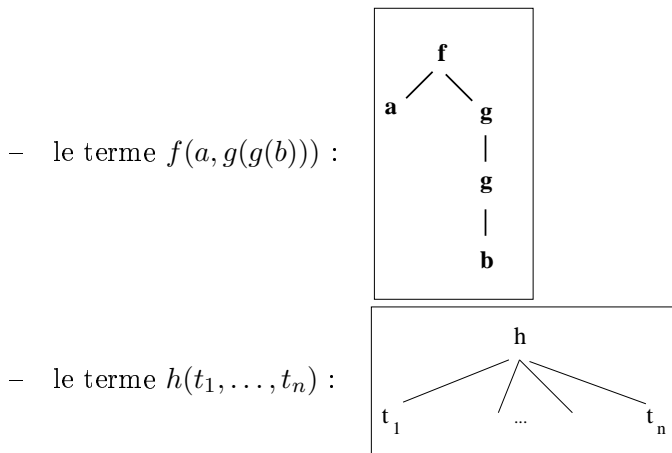
1. les variables sont des termes
2. si  $t_1, \dots, t_n$  sont des termes, et si  $f \in \mathcal{F}$  est tel que  $ar(f) = n$  alors  $f(t_1, \dots, t_n)$  est encore un terme.

Si  $t$  est un terme, nous noterons  $Var(t)$  l'ensemble des variables de  $t$ . Un terme  $t$  est dit **clos** s'il ne contient aucune variable. Un terme est dit **linéaire** si chaque variable de  $Var(t)$  a au plus une occurrence dans  $t$ .

**Exemple 1** Considérons l'alphabet  $\{a : 0, b : 0, p : 2, m : 2\}$  :

- $p(a, m(b, b)), m(m(a, b), p(b, p(a, b)))$  sont des termes
- $p(b), m(a, p(m(a)), b)$  ne sont pas des termes.

Un terme peut être représenté sous la forme d'un arbre étiqueté :



**Définition 3 (Ensemble des positions d'un terme)** L'ensemble des positions  $Pos(t)$  d'un terme  $t$  est un mot sur  $\mathbb{N}$  défini de la façon suivante :

1.  $Pos(t) = \epsilon$  si  $t$  est une variable. ( $\epsilon$  représente le mot vide)
2.  $Pos(f(t_1, \dots, t_n)) = \{\epsilon\} \cup \{i.p \mid 1 \leq i \leq n \text{ et } p \in Pos(t_i)\}$ .

Si  $t$  est un terme et  $u$  une position de  $t$ , le sous terme de  $t$  à la position  $u$  sera noté  $t|_u$

**Définition 4 (Contexte)** *Un contexte est un terme  $C[\ ]$  de  $\mathcal{T}(\mathcal{F} \cup \{\square\}, \mathcal{X})$  dans lequel le nouveau symbole de constante n'apparaît qu'une seule fois. Pour tout contexte  $C[\ ]$ , et tout terme  $t$ ,  $C[t]$  désigne le terme obtenu en remplaçant  $\square$  par  $t$  dans  $C[\ ]$ .*

**Définition 5 (Système de réécriture)** *Un système de réécriture est un couple  $(\mathcal{F}, \mathcal{R})$ <sup>4</sup> où  $\mathcal{F}$  est un alphabet et  $\mathcal{R}$  un ensemble de règles de la forme  $l \rightarrow r$  où  $l, r \in \mathcal{T}(\mathcal{F}, \mathcal{X})$  et  $l$  n'est pas une variable et  $Var(r) \subseteq Var(l)$ .*

Une règle de réécriture est dite linéaire à gauche (à droite) si le membre gauche (droit) est linéaire. Une règle est linéaire si elle est linéaire à droite et à gauche.

**Exemple 2** *Par exemple, dans les règles suivantes*

$$x \rightarrow Plus(x, 0) \tag{2.1}$$

$$Plus(x, 0) \rightarrow x \tag{2.2}$$

$$0 \rightarrow Plus(x, y) \tag{2.3}$$

$$Plus(x, -x) \rightarrow 0 \tag{2.4}$$

$$\tag{2.5}$$

2.1 et 2.3 ne sont pas des règles de réécriture, 2.4 n'est pas linéaire à gauche.

**Définition 6 ( $\mathcal{T}(\mathcal{F})$ -substitution)** *Soient  $\mathcal{T}(\mathcal{F})$  un ensemble de termes et  $\mathcal{X}$  un ensemble de variables, une  $\mathcal{T}(\mathcal{F})$ -substitution est une fonction de  $\mathcal{X}$  vers  $\mathcal{T}(\mathcal{F})$ .*

*Nous dirons qu'une substitution est valide si et seulement si elle est injective.*

Une substitution qui à  $x$  et  $y$  associe 0 et 1 sera notée  $[x \mapsto 0, y \mapsto 1]$

**Définition 7 (Réécriture)**  *$(\mathcal{F}, \mathcal{R})$  un système de réécriture,  $s$  et  $t$  deux termes de  $\mathcal{T}(\mathcal{F} \cup \mathcal{X})$ , nous dirons que  $s$  se **réécrit** en  $t$  et nous noterons  $s \rightarrow_{\mathcal{R}} t$  s'il existe*

- $l \rightarrow r$  une règle de réécriture appartenant à  $(\mathcal{F}, \mathcal{R})$
- $\sigma$  une  $\mathcal{T}(\mathcal{F})$ -substitution
- $p$  une position de  $Pos(t)$

*tel que  $l\sigma = s|_p$  et  $t = s[r\sigma]_p$*

Nous noterons  $\rightarrow_{\mathcal{R}}^*$  la fermeture réflexive et transitive de  $\mathcal{R}$ .

**Définition 8 (Forme normale)** *Soit  $t$  un terme de  $\mathcal{T}(\mathcal{F} \cup \mathcal{X})$ , nous dirons que  $t$  est **irréductible** ou **en forme normale** si  $\forall t' (t \rightarrow_{\mathcal{R}}^* t') \Rightarrow (t = t')$ . Un terme est dit **normalisable** si  $\exists t'$  irréductible tel que  $t \rightarrow_{\mathcal{R}}^* t'$ . Nous noterons  $\text{Irr}(\mathcal{R})$  l'ensemble des termes irréductibles de  $\mathcal{R}$*

**Définition 9 (Terme accessible)** *Soient  $(\mathcal{F}, \mathcal{R})$  un système de réécriture et  $\mathbf{E}$  un ensemble de termes, nous appelons  $\mathcal{R}^*(\mathbf{E})$  l'ensemble des termes accessibles de  $\mathbf{E} : \{t \in \mathcal{T}(\mathcal{F}) \mid \exists t' \in \mathbf{E} t' \rightarrow_{\mathcal{R}}^* t\}$*

<sup>4</sup>Nous noterons plus simplement  $\mathcal{R}$  lorsqu'il n'y aura pas d'ambiguïté sur l'alphabet considéré.

**Exemple 3** *Considérons le terme  $t = f(a, g(b))$  et  $\mathcal{R}$  un système de réécriture réduit à la règle  $f(x, y) \rightarrow f(y, g(x))$ . Le terme  $f(g(a), g(g(b)))$  est accessible par  $\mathcal{R}$  à partir de  $t$  mais pas  $f(g(a), g(b))$*

**Définition 10 (Sorte finitaire et infinitaire)** *Soit  $\mathcal{R}$  un système de réécriture et  $s$  une sorte.  $s$  est dite finitaire s'il n'existe pas d'ensemble infini de termes clos irréductibles de sorte  $s$  par  $\mathcal{R}$ . Sinon  $s$  est dite infinitaire.*

**Exemple 4** *Considérons :*

- les sortes *Mois* et *An*,
- l'ensemble des termes *Janvier, ..., Decembre, 0 ... 2003, ...*,
- et le système de réécriture

$$\text{MoisSuivant}(\text{Janvier}, x) \rightarrow (\text{Fevrier}, x)$$

⋮

$$\text{MoisSuivant}(\text{Decembre}, x) \rightarrow (\text{Janvier}, \text{Succ}(x))$$

*La sorte *Mois* est finitaire alors que la sorte *An* est infinitaire.*

### 2.1.1 Rappel sur les relations d'ordre

**Définition 11 (Ordre et pré-ordre)** *Une relation de pré-ordre est une relation binaire transitive et réflexive, une relation d'ordre est de plus anti-symétrique.*

Une relation d'ordre sur un domaine  $D$  est une relation totale ou *ordre total* lorsque tous les éléments du domaine sont comparables. On appelle ordre strict une relation transitive et anti-symétrique.

**Définition 12 (Ordre bien fondé)** *Une relation d'ordre stricte  $<$  est dite bien fondée ou noëthérienne si il n'existe pas de chaîne infinie décroissante pour  $<$ .*

Par exemple, la relation d'ordre habituelle sur  $\mathbb{N}$  est un ordre bien fondé alors que l'ordre sur  $\mathbb{Q}$  ne l'est pas.

**Définition 13 (Stabilité)** *Une relation d'ordre  $<$  est dite stable par instanciation si pour tout couple de termes  $(s, t)$  et pour toute substitution  $\sigma$*

$$s < t \Rightarrow s\sigma < t\sigma$$

**Définition 14 (Monotonie)** *Une relation d'ordre  $<$  est dite monotone si pour tout couple de termes  $(s, t)$  et pour tout symbole de fonction  $f$*

$$s < t \Rightarrow f(\dots, s, \dots) < f(\dots, t, \dots)$$

**Définition 15 (Ordre de réduction)** *Un ordre de réduction est un ordre stable et monotone.*

**Définition 16 (Terminaison)** *Une système de réécriture  $\mathcal{R}$  termine s'il n'existe pas de chaîne de termes infinie telle que  $t_0 \rightarrow_{\mathcal{R}} t_1 \dots \rightarrow_{\mathcal{R}} t_p \rightarrow_{\mathcal{R}} \dots$*

**Proposition 1** Une relation de réécriture sur  $\mathcal{T}(\mathcal{F}, \mathcal{X})$  qui termine induit un ordre bien fondé  $\prec_{\mathcal{R}}$  sur les termes.

**Définition 17 (Extension multi-ensemble)** Un multi-ensemble est un ensemble où chaque élément peut avoir une ou plusieurs occurrences. Pour un ordre  $\prec$ , sur un domaine  $D$ , nous définissons l'extension  $\preccurlyeq$  multi-ensemble de  $\preceq$  comme la fermeture réflexive et transitive de la relation suivante définie pour un multi-ensemble  $M$  :

$$M \cup \{x\} \preccurlyeq M \cup \{y_1, \dots, y_n\} \text{ si } \forall i \ x \preceq y_i$$

## 2.2 Automates d'arbres

Les automates d'arbres offrent un moyen de représenter des ensembles de termes. Un ensemble de termes reconnu par un automate d'arbre est appelé langage régulier. Nous rappelons tout d'abord les opérations usuelles sur les automates d'arbres en vue de montrer comment la représentation par automate d'un langage  $E$  peut être utilisée pour reconnaître une sur-approximation de  $\mathcal{R}^*(\mathbf{E})$ , l'ensemble des termes accessibles par un système de réécriture  $\mathcal{R}$ .

### 2.2.1 Automates d'arbres

**Définition 18 (Configurations)** Soit  $\mathcal{F}$  un alphabet et  $\mathcal{Q}$  un ensemble d'états, l'ensemble des configurations  $\mathcal{T}(\mathcal{F} \cup \mathcal{Q})$  est le plus petit ensemble tel que :

les états sont des configurations,

si  $c_1, \dots, c_n$  sont des configurations, et si  $f \in \mathcal{F}$  est tel que  $ar(f) = n$  alors  $f(c_1, \dots, c_n)$  est encore une configuration.

**Définition 19 (Automate d'arbre)** Un automate d'arbre est un 4-uple  $\mathcal{A} = \langle \mathcal{F}, \mathcal{Q}, \mathcal{Q}_f, \Delta \rangle$  où

$\mathcal{F}$  est un alphabet,

$\mathcal{Q}$  un ensemble d'états,

$\mathcal{Q}_f$  un ensemble d'états finaux,

$\Delta$  un ensemble de transitions normalisées de la forme  $f(q_1, \dots, q_n) \rightarrow q$  où

$$f \in \mathcal{F} \text{ et } ar(f) = n$$

$$q_1, \dots, q_n, q \in \mathcal{Q}$$

$\Delta$  est aussi appelé table de transition.

Nous noterons  $\rightarrow_{\Delta}^*$  la relation de réécriture induite par  $\Delta$

Un automate est *déterministe* s'il existe pas deux transitions  $c_1 \rightarrow q_1$  et  $c_2 \rightarrow q_2$  telles que  $c_1 = c_2$  et  $q_1 \neq q_2$

**Définition 20 (Taille d'un automate)** La taille d'un automate  $\mathcal{A} = \langle \mathcal{F}, \mathcal{Q}, \mathcal{Q}_f, \Delta \rangle$  est donnée par la somme des tailles de ses composants :

La taille d'une règle de transition  $\|f(q_1, \dots, q_n) \rightarrow q\|$  est  $n + 1$ . La taille  $\|\Delta\|$  de la table de transition est la somme des tailles des règles de transitions. La taille de l'automate est définie par  $\|\mathcal{A}\| = |\mathcal{Q}| + \|\Delta\|$ .

Intuitivement, la taille d'un automate correspond au nombre de symboles utilisés pour le décrire.

**Définition 21 (terme reconnu par un automate)** Soit  $t$  un terme  $\in \mathcal{T}(\mathcal{F})$ ,  $\mathcal{A} = \langle \mathcal{F}, \mathcal{Q}, \mathcal{Q}_f, \Delta \rangle$  un automate et  $q \in \mathcal{Q}$ .

- $t$  est reconnu par  $q$  si et seulement si  $t \xrightarrow{\star} \Delta q$
- $t$  est reconnu par  $\mathcal{A}$  si et seulement si il existe un état final  $q$  tel que  $t$  est reconnu par  $q$
- le langage reconnu par  $\mathcal{A}$  est l'ensemble des termes reconnus par  $\mathcal{A}$  noté  $\mathcal{L}(\mathcal{A})$ .

**Exemple 5** Considérons l'automate  $\mathcal{A}$  suivant :

<b>Automaton A</b>	
<b>States</b>	
$q1 \dots q4$	
<b>Final States</b>	
$q1$	
<b>Transitions</b>	
$a$	$\rightarrow q1$
$b$	$\rightarrow q2$
$g(q1, q2, q2)$	$\rightarrow q3$
$f(q1, q3)$	$\rightarrow q1$

Cet automate reconnaît les termes  $a, f(a, g(a, b, b))$  mais pas les termes  $g(b, b, b), f(a, a) \dots$

### 2.2.2 Opérations usuelles

Les principales opérations sur les automates d'arbres sont les opérations ensemblistes sur les langages telles que l'union, l'intersection, le complémentaire.

**Proposition 2** La classe des langages réguliers est close pour les opérations ensemblistes.

Ce qui signifie en particulier que pour deux langages réguliers  $E$  et  $F$ , nous savons écrire des automates qui reconnaissent  $E \cup F, E \cap F$  ou  $\overline{E} = \mathcal{T}(\mathcal{F}) \setminus E$ . Pour les définitions qui vont suivre, nous considérons deux automates d'arbres  $\mathcal{A} = \langle \mathcal{F}, \mathcal{Q}_A, \mathcal{Q}_{A,f}, \Delta_A \rangle$  et  $\mathcal{B} = \langle \mathcal{F}, \mathcal{Q}_B, \mathcal{Q}_{B,f}, \Delta_B \rangle$ , nous ne proposons ici que les opérations d'union et d'intersection qui nous intéressent pour la suite. Le lecteur trouvera une documentation plus riche sur les automates d'arbres dans [CDG<sup>+</sup>97].

**Définition 22 (Union d'automates)** L'automate  $\mathcal{C}$  défini par

- $\mathcal{F}$
- $\mathcal{Q}_A \times \mathcal{Q}_B$
- $\mathcal{Q}_{A,f} \times \mathcal{Q}_B \cup \mathcal{Q}_A \times \mathcal{Q}_{B,f}$
- $\{f((q_1, q'_1), \dots, (q_n, q'_n)) \rightarrow (q, q')\}$  tel que  $f(q_1, \dots, q_n) \rightarrow q \in \Delta_A$  et  $f(q'_1, \dots, q'_n) \rightarrow q' \in \Delta_B$

reconnaît le langage  $\mathcal{L}(\mathcal{A}) \cup \mathcal{L}(\mathcal{B})$  :

**Définition 23 (Intersection d'automates)** L'automate  $\mathcal{D}$  défini par

- $\mathcal{F}$
- $\mathcal{Q}_A \times \mathcal{Q}_B$

- $\mathcal{Q}_{\mathcal{A},f} \times \mathcal{Q}_{\mathcal{B},f}$
- $\{f((q_1, q'_1), \dots, (q_n, q'_n)) \rightarrow (q, q')\}$  tel que  $f(q_1, \dots, q_n) \rightarrow q \in \Delta_{\mathcal{A}}$  et  $f(q'_1, \dots, q'_n) \rightarrow q' \in \Delta_{\mathcal{B}}$

reconnaît le langage  $\mathcal{L}(\mathcal{A}) \cap \mathcal{L}(\mathcal{B})$  :

**Propriété: 1** Soit deux automates  $\mathcal{A} = \langle \mathcal{F}, \mathcal{Q}_{\mathcal{A}}, \mathcal{Q}_{\mathcal{A},f}, \Delta_{\mathcal{A}} \rangle$  et  $\mathcal{B} = \langle \mathcal{F}, \mathcal{Q}_{\mathcal{B}}, \mathcal{Q}_{\mathcal{B},f}, \Delta_{\mathcal{B}} \rangle$ , la taille des automates  $\|\mathcal{C}\|$  et  $\|\mathcal{D}\|$  définis précédemment est dans  $\mathcal{O}(\|\mathcal{A}\| \cdot \|\mathcal{B}\|)$

Il nous reste à définir l'ensembles des états *accessibles* et *utiles* :

- Un état est accessible s'il reconnaît au moins un terme.
- Un état est utile s'il apparaît dans au moins une séquence de réécriture menant à un état final.

A partir de là nous pouvons donner une procédure suffisante pour décider le vide d'un automate : le langage reconnu par un automate est vide si et seulement si aucun état final n'est accessible.

Ces opérations sont utilisables dans *Timbuk* [GVTT01] qui a été réalisé en collaboration avec Thomas Genet *Caml*[LDG<sup>+</sup>00] et distribué sous licence Gnu Library General Public License. A partir de maintenant nous décrirons aussi bien nos automates dans la syntaxe *Timbuk* , par exemple un automate  $\mathcal{A}$  dont l'alphabet contient les constantes  $a$  et  $b$ , le symbole unaire  $g$  et le symbole binaire  $f$  et décrit par la syntaxe usuelle :  $\mathcal{A} = \langle \mathcal{Q} = \{q_a, q_b, q_g, q_f\}, \mathcal{Q}_f = \{q_f\}, \Delta = a \rightarrow q_a, b \rightarrow q_b, g(q_b) \rightarrow q_g, f(q_a, q_g) \rightarrow q_f \rangle$  pourra dorénavant être décrit par

**Automaton A**

**States**

$q_a \ q_b \ q_g \ q_f$

**Final States**

$q_f$

**Transitions**

$a \rightarrow q_a$

$b \rightarrow q_b$

$g(q_b) \rightarrow q_g$

$f(q_a, q_g) \rightarrow q_f$



# 3

## Complétion d'automates

## Sommaire

<b>3.1</b>	<b>Compléter <math>\Delta</math> par <math>\mathcal{R}</math></b> . . . . .	<b>20</b>
3.1.1	Normalisations de transitions . . . . .	22
<b>3.2</b>	<b>Automate clos pour un système de réécriture</b> . . . . .	<b>24</b>
3.2.1	Condition de linéarité . . . . .	24
3.2.2	Condition pour que $\mathcal{A}_{\mathcal{R},\alpha}^*$ existe toujours . . . . .	27
<b>3.3</b>	<b>Filtrage dans un automate</b> . . . . .	<b>28</b>
3.3.1	Algorithmes existants . . . . .	28
3.3.2	Algorithme proposé . . . . .	29
3.3.3	Comparaison des algorithmes, complexité du problème . . . . .	34
<b>3.4</b>	<b>Rôle de l'approximation</b> . . . . .	<b>34</b>
3.4.1	Choix de la fonction d'abstraction . . . . .	35
3.4.2	Calcul exact des descendants . . . . .	37
<b>3.5</b>	<b>Utiliser la complétion d'automates pour des preuves de propriétés</b>	<b>39</b>

Considérons  $E$  un ensemble régulier de termes,  $\mathcal{R}$  un système de réécriture, l'ensemble des termes accessibles à partir de  $E$  par  $\mathcal{R}$  noté  $\mathcal{R}^*(E)$ , est l'ensemble  $\{t \mid \exists t' \in E \ t' \rightarrow_{\mathcal{R}}^* t\}$ . Nous nous intéressons ici à un algorithme présenté initialement dans [Gen98] permettant de calculer une sur-approximation de  $\mathcal{R}^*(E)$  grâce à une complétion de  $\mathcal{A}$  (l'automate reconnaissant  $E$ ) par  $\mathcal{R}$ . Ce procédé s'inspire de l'algorithme de complétion de Knuth Bendix qui considère un système de réécriture et vise à le rendre confluent <sup>5</sup> en examinant tous les recouvrements possibles entre les règles et en rajoutant les règles nécessaires.

### 3.1 Compléter $\Delta$ par $\mathcal{R}$

Soit  $\mathcal{A} = \langle \mathcal{F}, \mathcal{Q}, \mathcal{Q}_f, \Delta \rangle$  un automate et  $\mathcal{R}$  un système de réécriture, nous allons chercher à *étendre* la relation  $\rightarrow_{\Delta}$ , ce qui définira un nouvel automate  $\mathcal{A}_{\mathcal{R},\alpha}^*$  vérifiant, pour tout couple de termes  $(t, u)$ , si  $t$  est reconnu par  $\mathcal{A}$  et que  $t \rightarrow_{\mathcal{R}}^* u$  alors  $u$  est reconnu par  $\mathcal{A}_{\mathcal{R},\alpha}^*$  autrement dit  $\mathcal{A}_{\mathcal{R},\alpha}^*$  reconnaît un sur-ensemble de  $\mathcal{R}^*(\mathcal{L}(\mathcal{A}))$ .

Soit  $\mathcal{A} = \langle \mathcal{F}, \mathcal{Q}, \mathcal{Q}_f, \Delta \rangle$  un automate,  $\mathcal{R}$  un système de réécriture. Pour toute règle  $l \rightarrow r$  et toute  $\mathcal{Q}$ -substitution  $\sigma$ <sup>6</sup>, nous allons chercher les paires critiques  $(r\sigma, q)$  telles que  $l\sigma \rightarrow_{\mathcal{R}}^* r\sigma$ ,  $l\sigma \rightarrow_{\Delta}^* q$  et  $r\sigma \not\rightarrow_{\Delta}^* q$ . Dans la suite nous détaillerons la recherche des paires critiques et proposerons un algorithme basé sur l'intersection d'automates. Si la transition  $r\sigma \rightarrow_{\Delta}^* q$  est déjà normalisée, nous l'ajoutons à  $\Delta$ , sinon il faudra normaliser la transition avant de l'ajouter. L'exemple qui suit présente un cas simple où la transition à ajouter est déjà normalisée :

**Exemple 6** *Considérons la règle de réécriture  $a \rightarrow b$  et l'automate  $\mathcal{A}$  défini par*

<sup>5</sup> $\mathcal{R}$  est un système confluent si pour tous termes  $t, u, v$  vérifiant  $t \rightarrow_{\mathcal{R}}^* u$  et  $t \rightarrow_{\mathcal{R}}^* v$  il existe un terme  $w$  tel que  $u \rightarrow_{\mathcal{R}}^* w$  et  $v \rightarrow_{\mathcal{R}}^* w$ .

<sup>6</sup>voir la définition 28.

<b>Automaton A</b>
<b>States</b>
$qf$
<b>Final States</b>
$qf$
<b>Transitions</b>
$a \rightarrow qf$

Le terme  $a$  est reconnu par  $\mathcal{A}$ ,  $a$  se réécrit en  $b$  mais  $b$  n'est pas reconnu par l'automate. Compléter  $\Delta$  par  $\mathcal{R}$  consiste à enrichir l'automate de sorte qu'il reconnaisse aussi le terme  $b$ . Il suffit d'ajouter la transition  $b \rightarrow qf$ . Nous obtenons ensuite un automate sur lequel nous ne pouvons plus appliquer de règle de réécriture pour obtenir de nouveaux termes. La complétion s'achève.

Le premier problème qui va se poser est celui des transitions non normalisées. Considérons la règle de réécriture  $a \rightarrow f(a, a)$  et l'automate précédent, le terme  $a$  est reconnu par  $qf$  dans  $\mathcal{A}$ ,  $a$  se réécrit en  $f(a, a)$  qui n'est pas reconnu par l'automate. Le principe de la complétion est calculer un nouvel automate contenant  $\mathcal{A}$  et dont la table de transition  $\Delta'$  contient une ou plusieurs nouvelles transitions permettant de reconnaître le terme  $f(a, a)$ . Le premier problème est que nous ne pouvons pas ajouter directement la règle  $f(a, a) \rightarrow qf$ , car ce n'est pas une transition d'automate. Nous allons proposer de remplacer cette règle par un ensemble de règles d'automates reconnaissant au moins le terme  $f(a, a)$ . Le second problème sera la terminaison du processus de complétion, jusqu'ici nous n'avons présenté que des exemples sur lesquels la complétion termine naturellement, ce qui n'est bien entendu pas le cas en général :

**Exemple 7** Considérons le système de réécriture réduit à la règle  $Succ(x) \rightarrow Succ(Succ(x))$  et l'automate décrit par

<b>Automaton A</b>
<b>States</b>
$q \ qf$
<b>Final States</b>
$qf$
<b>Transitions</b>
$0 \rightarrow q$
$Succ(q) \rightarrow qf$

L'état  $qf$  reconnaît le terme  $Succ(0)$  et celui ci se réécrit en  $Succ(Succ(0))$ . La première étape est d'enrichir la table de transition pour que  $qf$  reconnaisse le terme  $Succ(Succ(0))$ .

Nous choisissons de remplacer la règle  $Succ(Succ(0)) \rightarrow qf$  par l'ensemble des transitions :

$$\delta = \{0 \rightarrow q_0, Succ(q_0) \rightarrow q_1, Succ(q_1) \rightarrow qf\}$$

Nous enrichissons  $\Delta$  à l'aide de  $\delta$  et itérons le processus sur l'automate enrichi, la règle de réécriture peut être appliquée une seconde fois pour produire la transition  $Succ(Succ(Succ(0))) \rightarrow qf$ . Le processus ne termine pas.

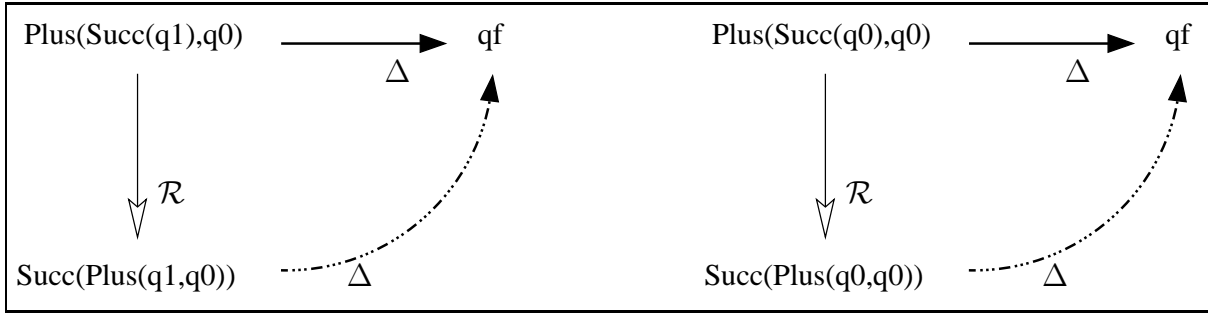


FIG. 3.1 – paires critiques

Dans la suite de ce chapitre nous formaliserons la complétion d'automate, montrerons comment normaliser les transitions à ajouter et comment la façon de normaliser les transitions peut forcer le processus de complétion à s'achever.

### 3.1.1 Normalisations de transitions

Les transitions à ajouter vont être normalisées à l'aide d'états dont le choix sera guidé par une fonction d'abstraction.

**Définition 24 (abstraction)** Soient  $\mathcal{Q}$  un ensemble d'états,  $E$  un ensemble de termes de  $\mathcal{T}(\mathcal{F}, \mathcal{Q})$ , une abstraction de  $E$  est une application  $\alpha : E \rightarrow \mathcal{Q}$  telle que pour tout état  $q$ ,  $\alpha(q) = q$ .

**Définition 25 (normalisation des transitions)** Soit  $\mathcal{F}$  un alphabet,  $\mathcal{Q}$  un ensemble d'états, et  $c \rightarrow q$  une règle de réécriture telle que  $c \in \mathcal{T}(\mathcal{F}, \mathcal{Q})$  et  $\alpha$  une fonction d'abstraction de  $\mathcal{T}(\mathcal{F}, \mathcal{Q})$ . La normalisation de  $c \rightarrow q$  suivant  $\alpha$  est l'ensemble noté  $Norm_\alpha(c \rightarrow q)$  et est défini par induction :

- $Norm_\alpha(c \rightarrow q) = \{c \rightarrow q\}$  si  $c$  est soit une constante, soit de la forme  $f(q_1, \dots, q_n)$
- $Norm_\alpha(q \rightarrow q) = \emptyset$  pour  $q \in \mathcal{Q}$
- $Norm_\alpha(f(c_1, \dots, c_n) \rightarrow q) = \{f(\alpha(c_1), \dots, \alpha(c_n)) \rightarrow q\} \cup \bigcup_{i \in \{1, \dots, n\}} Norm_\alpha(c_i \rightarrow \alpha(c_i))$

**Exemple 8** Considérons

- l'alphabet  $\{Plus : 2, Succ : 1, 0 : 0\}$
- l'ensemble d'états  $\{q_0, q_1, q_f, q_{new1}\}$
- la fonction d'abstraction  $\alpha : \mathcal{T}(\mathcal{F} \cup \mathcal{Q}) \rightarrow \mathcal{Q}$  définit partiellement par  $\alpha(Plus(q_1, q_0)) = q_{new1}$

$$Norm_\alpha(Succ(Plus(q_1, q_0)) \rightarrow q_f) = \{Succ(q_{new1}) \rightarrow q_f, Plus(q_1, q_0) \rightarrow q_{new1}\}$$

Nous avons maintenant suffisamment d'outils pour définir une étape de complétion d'un automate  $\mathcal{A}$  par un système de réécriture  $\mathcal{R}$  :

**Définition 26 (Complétion d'automate)** Soit  $\mathcal{A} = \langle \mathcal{F}, \mathcal{Q}, \mathcal{Q}_f, \Delta \rangle$  un automate,  $\mathcal{R}$  un système de réécriture linéaire à gauche, et  $\alpha : \mathcal{T}(\mathcal{F} \cup \mathcal{Q}) \rightarrow \mathcal{Q}$  une abstraction de  $\mathcal{T}(\mathcal{F} \cup \mathcal{Q})$ . L'automate obtenu après une étape de complétion de  $\mathcal{A}$  pour  $\mathcal{R}$  et suivant  $\alpha$  est défini par :

$$\mathcal{A}_{\mathcal{R}, \alpha}^1 = \langle \mathcal{F}, \mathcal{Q}, \mathcal{Q}_f, \Delta \cup \bigcup_{t \xrightarrow{\Delta}^* q, t \xrightarrow{\mathcal{R}} v} \{Norm_\alpha(v \rightarrow q)\} \rangle$$

**Exemple 9** Reprenons l'alphabet, l'ensemble d'états de l'exemple précédent et la fonction d'abstraction  $\alpha$  enrichie par  $\alpha(Plus(q_0, q_0)) = q_{new2}$  et considérons l'automate suivant qui reconnaît l'ensemble des termes de la forme  $Plus(Succ^n(0), 0)$  ( $n > 0$ ).

$$\mathcal{Q} = \{q_0, q_1, q_f\}$$

$$\mathcal{Q}_f = \{q_f\}$$

$$\Delta = \{$$

$$0 \rightarrow q_0$$

$$Succ(q_0) \rightarrow q_1$$

$$Succ(q_1) \rightarrow q_1$$

$$Plus(q_1, q_0) \rightarrow q_f \} \}$$

Le système de réécriture  $\mathcal{R}$  définit la fonction *Plus* :

$$Plus(0, x) \rightarrow_{\mathcal{R}} x$$

$$Plus(Succ(x), y) \rightarrow_{\mathcal{R}} Succ(Plus(x, y))$$

Nous cherchons les paires critiques, c'est à dire les couples  $(r\sigma, q)$  tels que  $l\sigma \rightarrow_{\mathcal{R}}^* r\sigma$ ,  $l\sigma \rightarrow_{\Delta}^* q$  et  $r\sigma \not\rightarrow_{\Delta}^* q$ . (cf. fig 3.1).

Nous trouvons  $Succ(Plus(q_1, q_0)) \rightarrow q_f$  et  $Succ(Plus(q_0, q_0)) \rightarrow q_f$ , d'après la définition précédente, l'automate obtenu après une étape de complétion par  $\mathcal{R}$  et suivant  $\alpha$  est :

<b>Automaton</b>	<i>Terme</i>
<b>States</b>	
$q_0 \ q_1 \ q_f$	
<b>Final States</b>	
$q_f$	
<b>Transitions</b>	
$0 \rightarrow q_0$	
$Succ(q_0) \rightarrow q_1$	
$Succ(q_1) \rightarrow q_1$	
$Plus(q_1, q_0) \rightarrow q_f$	
$Succ(q_{new1}) \rightarrow q_f$	
$Plus(q_1, q_0) \rightarrow q_{new1}$	
$Succ(q_{new2}) \rightarrow q_f$	
$Plus(q_0, q_0) \rightarrow q_{new2}$	

Le langage de l'automate obtenu après une étape de complétion contient le langage de l'automate initial ainsi que les termes  $Succ(Plus(0, 0))$  et  $Succ(Plus(Succ^n(0), 0))$

Nous écrivons  $\mathcal{A}_{\mathcal{R}, \alpha}^n \vdash_{\mathcal{R}} \mathcal{A}_{\mathcal{R}, \alpha}^{n+1}$ , si l'automate  $\mathcal{A}_{\mathcal{R}, \alpha}^{n+1}$  peut être obtenu à partir de  $\mathcal{A}_{\mathcal{R}, \alpha}^n$  en une étape de complétion. Etant donné une séquence  $\mathcal{A} \vdash_{\mathcal{R}} \mathcal{A}_{\mathcal{R}, \alpha}^1, \dots, \mathcal{A}_{\mathcal{R}, \alpha}^n \vdash_{\mathcal{R}} \mathcal{A}_{\mathcal{R}, \alpha}^{n+1}, \dots$ , nous définissons  $\mathcal{A}_{\mathcal{R}, \alpha}^*$  appelé automate limite de  $\mathcal{A}$  obtenu par complétion de  $\mathcal{A}$  par  $\mathcal{R}$  :

$$\mathcal{A}_{\mathcal{R}, \alpha}^* = \langle \mathcal{F}, \mathcal{Q}, \mathcal{Q}_f, \Delta \cup \bigcup_{n \rightarrow \text{infy}} \Delta^i \rangle$$

Nous dirons qu'une séquence de complétion est réussie si et seulement si elle est finie et dans ce cas il existe  $N_0 \in \mathbb{N}$  tel que  $\mathcal{A}_{\mathcal{R},\alpha}^* = \mathcal{A}_{\mathcal{R},\alpha}^{N_0}$ . Trivialement  $\mathcal{L}(\mathcal{A}_{\mathcal{R},\alpha}^n) \subset \mathcal{L}(\mathcal{A}_{\mathcal{R},\alpha}^{n+1})$ . Nous montrons tout d'abord que si  $\mathcal{A}_{\mathcal{R},\alpha}^*$  existe alors c'est un automate clos par  $\mathcal{R}$  puis nous montrerons que sous certaines hypothèses sur  $\alpha$ , l'automate  $\mathcal{A}_{\mathcal{R},\alpha}^*$  existe toujours.

## 3.2 Automate clos pour un système de réécriture

Intuitivement un automate  $\mathcal{A}$  est clos pour un système de réécriture  $\mathcal{R}$  si pour tout terme  $t$  qu'il reconnaît,  $\mathcal{A}$  reconnaît aussi tous les termes accessibles par  $\mathcal{R}$  à partir de  $t$ .

**Définition 27 (Automate clos pour un système de réécriture)** *Soit  $\mathcal{R}$  un système de réécriture et  $\mathcal{A} = \langle \mathcal{F}, \mathcal{Q}, \mathcal{Q}_f, \Delta \rangle$  un automate d'arbre. Le langage  $\mathcal{L}(\mathcal{A})$  est clos pour  $\mathcal{R}$  si et seulement si pour tout terme  $t$  reconnu par un état  $q \in \mathcal{Q}$  si  $t \rightarrow_{\mathcal{R}} t'$  alors  $t' \rightarrow_{\Delta}^* q$ .*

**Proposition 3** *Soit  $\mathcal{R}$  un système de réécriture et un automate  $\mathcal{A}$ , si  $\mathcal{A}$  est clos par  $\mathcal{R}$  alors  $\mathcal{R}^*(\mathcal{L}(\mathcal{A})) \subset \mathcal{L}(\mathcal{A})$*

**Démonstration:**  $\triangleright$

Soit  $t \in \mathcal{L}(\mathcal{A})$  tel que  $t \rightarrow_{\mathcal{R}} t'$ , nous savons qu'il existe un état final  $q$  de  $\mathcal{A}$  tel que  $t \rightarrow_{\Delta}^* q$ , comme par ailleurs  $\mathcal{A}$  est clos par  $\mathcal{R}$  nécessairement  $t' \rightarrow_{\Delta}^* q$  et donc  $t' \in \mathcal{L}(\mathcal{A})$ .  $\triangleleft$

### 3.2.1 Condition de linéarité

Dans cette partie nous étendons le calcul aux systèmes de réécriture non linéaires. Pour un automate  $\mathcal{A}$  et un système de réécriture  $\mathcal{R}$ , nous définirons la condition de linéarité induite par  $\mathcal{R}$  sur  $\mathcal{A}$ .

Effectuer une étape de complétion d'un automate  $\mathcal{A}$  par un système de réécriture  $\mathcal{R}$ , c'est rechercher les termes  $t$  tels que

- $t$  est une instance d'un membre gauche de règle  $l \rightarrow r$  de  $\mathcal{R}$ ,
- $t$  est reconnu par un état  $q$  de  $\mathcal{A}$ .

Ce travail de recherche de paires critiques se fait dans l'automate, comme décrit en 3.1. Nous cherchons les  $\mathcal{Q}$ -substitutions  $\theta$  telles que  $l\theta \rightarrow_{\Delta}^* q$ . Si le membre gauche de règle  $l$  dont nous cherchons les instances dans  $\mathcal{A}$  n'est pas linéaire alors il est possible d'oublier des substitutions :

**Exemple 10** *Considérons l'automate suivant :*

$$\mathcal{A} = \left\langle \begin{array}{l} \mathcal{F} = \{a : 0, g : 1, f : 2\} \\ \mathcal{Q} = \{q_{a1}, q_{a2}, q_f\} \\ \mathcal{Q}_f = \{q_f\} \\ \Delta = a \rightarrow_{\Delta} q_{a1}, a \rightarrow_{\Delta} q_{a2}, f(q_{a1}, q_{a2}) \rightarrow_{\Delta} q_f \end{array} \right\rangle$$

*et le système de réécriture  $\mathcal{R}$  réduit à la seule règle  $f(x, x) \rightarrow g(x)$ .*

*Il n'existe aucune  $\mathcal{Q}$ -substitution  $\tau$  telle que  $f(x, x)\tau$  soit reconnu par un état de  $\mathcal{A}$  puisque qu'il est impossible de filtrer  $f(x, x)$  sur  $f(q_{a1}, q_{a2})$ . Considérons maintenant le terme  $l' = f(x_1, x_2)$  et l'ensemble des contraintes  $\{x_1 = x_2\}$  et la  $\mathcal{Q}$ -substitution  $\sigma = [x_1 \leftarrow q_{a1}, x_2 \leftarrow q_{a2}]$ ,  $l'\sigma$  est reconnu par  $\mathcal{A}$  et  $a \rightarrow_{\Delta}^* q_{a1}$  et  $a \rightarrow_{\Delta}^* q_{a2}$ . La condition de linéarité induite par  $f(x, x) \rightarrow_{\mathcal{R}} g(x)$  n'est pas vérifiée par  $\mathcal{A}$ .*

*Dans cet exemple  $\mathcal{R}^*(\mathcal{L}(\mathcal{A}))$  n'est pas inclus dans le langage reconnu par  $\mathcal{A}_{\mathcal{R},\alpha}^* = \mathcal{A}$ , l'automate n'est pas clos par  $\mathcal{R}$ .*

Nous pourrions envisager de modifier le processus de complétion et d'inclure une étape de détermination de l'automate après chaque étape de complétion mais cela pose des problèmes de complexité, pour un automate ayant  $q$  états, l'automate déterministe qui lui est équivalent contient  $2^q$  états [CDG<sup>+</sup>97]. Intuitivement, si  $l = f(x, x)$ , nous définissons le terme  $l'$  obtenu par renommage de  $x$   $l' = f(x_1, x_2)$  et l'ensemble de contraintes  $\{x_1 = x_2\}$ . La condition de linéarité induite par  $\mathcal{R}$  sur  $\mathcal{A}$  est vérifiée si pour toute  $\mathcal{Q}$ -substitution  $\sigma = [x_1 \leftarrow q_1, x_2 \leftarrow q_2]$  telle que  $l'\sigma$  est reconnu par  $\mathcal{A}$ , il n'existe aucun terme  $t$  tel que  $t \rightarrow_{\Delta}^* q_1$  et  $t \rightarrow_{\Delta}^* q_2$ .

**Définition 28 ( $\mathcal{Q}$ -substitution)** *Soit  $\mathcal{Q}$  un ensemble d'états,  $\mathcal{X}$  un ensemble de variables, une  $\mathcal{Q}$ -substitution est une application partielle de  $\mathcal{X}$  vers  $\mathcal{Q}$ .*

**Définition 29 (Union de  $\mathcal{Q}$ -substitutions)** *Soit  $\sigma$  et  $\theta$  deux substitutions de domaines inclus dans l'ensemble des variables  $\mathcal{X}$ , la substitution  $\sigma \circ \tau$  ne peut pas être définie comme  $\sigma(\tau(x))$  puisque  $\tau(x)$  est un état et que le domaine de  $\sigma$  est un ensemble de variables, nous définissons donc l'union de deux  $\mathcal{Q}$  substitutions de la manière suivante :*

$$\begin{aligned} \sigma \circ \tau(x) &= \tau(x) && \text{si } \tau(x) = \sigma(x) \text{ ou si } \sigma(x) \text{ n'est pas défini} \\ \sigma \circ \tau(x) &= \sigma(x) && \text{si } \tau(x) \text{ n'est pas défini} \\ \sigma \circ \tau &&& \text{n'est pas défini sinon.} \end{aligned}$$

Soit  $\mathcal{A} = \langle \mathcal{F}, \mathcal{Q}, \mathcal{Q}_f, \Delta \rangle$  un automate et  $l \rightarrow r$  une règle de réécriture,  $\{x_1, \dots, x_n\}$  l'ensemble des variables ayant plus d'une occurrence dans  $l$  et  $\mathcal{Y}$  un ensemble de variables distinct de  $\mathcal{X}$ . Soit  $(l', E)$  un terme et un ensemble de contraintes obtenu par renommage des variables non linéaires de  $l$ . Nous définissons la fonction  $\mathcal{R}en$  telle que  $\mathcal{R}en(l) = (l', E)$  définie par récurrence structurelle par :

- si  $l$  est une constante ou une variable  $x$  telle que  $x \notin \{x_1, \dots, x_n\}$  alors  $\mathcal{R}en(l) = (l, \emptyset)$
- si  $l$  est une variable  $x_i \in \{x_1, \dots, x_n\}$  alors  $\mathcal{R}en(l) = (l[x_i \leftarrow y_j], \{x_i = y_j\})$  où les  $y_j \in \mathcal{Y}$
- si  $l$  est de la forme  $f(t_1, \dots, t_n)$  et que  $(t'_i, E_i) = \mathcal{R}en(t_i)$  pour  $1 \leq i \leq n$  alors  $\mathcal{R}en(l) = (f(t'_1, \dots, t'_n), \bigcup_i E_i)$ .

Pour tout couple de variable  $(y, y')$  de  $\mathcal{Y}$ , nous dirons que l'équation  $y = y'$  est déductible de  $E$  si il existe dans  $E$  des équations  $y = y_1, y_1 = y_2, \dots, y_n = y'$ .

Pour un automate  $\mathcal{A} = \langle \mathcal{F}, \mathcal{Q}, \mathcal{Q}_f, \Delta \rangle$  et une règle  $l \rightarrow r$ , la condition de linéarité est vérifiée si pour toute équation  $y_i = y_j$  déductible de l'ensemble des contraintes  $E$  défini par  $\mathcal{R}en(l) = (l', E)$ , et pour toute  $\mathcal{Q}$ -substitution  $\sigma$  telle que  $\sigma(y_i) = q_i$  et  $\sigma(y_j) = q_j$ , il n'existe aucun terme  $t$  vérifiant  $t \rightarrow_{\Delta}^* q_i$  et  $t \rightarrow_{\Delta}^* q_j$ .

En pratique, pour chaque équation  $y_i = y_j$  déductible de l'ensemble des contraintes, et pour toute substitution  $\sigma$  telle qu'il existe deux états  $\sigma(y_i) = q_i$  et  $\sigma(y_j) = q_j$ , pour savoir si la condition de linéarité induite par la contrainte  $y_i = y_j$  est vérifiée, il suffit de calculer l'intersection des automates  $\mathcal{A}_i = \langle \mathcal{F}, \mathcal{Q}, \{q_i\}, \Delta \rangle$  et  $\mathcal{A}_j = \langle \mathcal{F}, \mathcal{Q}, \{q_j\}, \Delta \rangle$ . Si le langage reconnu par l'automate intersection est vide alors il n'existe aucun terme de  $\mathcal{A}$  reconnu à la fois par  $q_i$  et  $q_j$ .

**Définition 30 (Condition de linéarité)** *Soient  $\mathcal{A} = \langle \mathcal{F}, \mathcal{Q}, \mathcal{Q}_f, \Delta \rangle$  un automate et  $l \rightarrow r$  une règle de réécriture telle que  $\mathcal{R}en(l) = (l', E)$  et  $l'\sigma'$  est reconnu par un état de  $\mathcal{A}$ .*

*Nous dirons que  $\mathcal{A}$  respecte la condition de linéarité induite par  $l \rightarrow r$ , si pour chaque équation  $y_i = y_j$  déductible de l'ensemble des contraintes  $E$  et pour toute  $\mathcal{Q}$ -substitution  $\sigma'$  :*

Si il existe deux états distincts  $q_i$  et  $q_j$  vérifiant  $\sigma'(y_i) = q_i$  et  $\sigma'(y_j) = q_j$   
alors qu'il n'existe aucun terme  $t$  vérifiant  $t \rightarrow_{\Delta}^* q_i$  et  $t \rightarrow_{\Delta}^* q_j$ .

En particulier, les automates déterministes vérifient la condition de linéarité induite par un système de réécriture  $\mathcal{R}$ .

**THEOREME 1: Condition de linéarité**

Soit  $\mathcal{A} = \langle \mathcal{F}, \mathcal{Q}, \mathcal{Q}_f, \Delta \rangle$  un automate,  $\mathcal{R}$  un système de réécriture, si  $\mathcal{A}$  respecte la condition de linéarité induite par  $\mathcal{R}$  alors pour tout terme  $t$  reconnu par un état  $q$  de l'automate : si il existe une substitution  $\rho$  et une règle  $l \rightarrow r$  telle que  $t = l\rho$  et  $l\rho \rightarrow_{\Delta}^* q$  alors il correspond une  $\mathcal{Q}$ -substitution  $\sigma$  telle que  $t \rightarrow_{\Delta}^* l\sigma$  et  $l\sigma \rightarrow_{\Delta}^* q$ .

**Démonstration:** ▷

Considérons un terme  $t$  et une règle  $l \rightarrow r$  telle qu'il existe une substitution  $\rho$  vérifiant  $t = l\rho$  avec  $\rho : \mathcal{X} \mapsto \mathcal{T}(\mathcal{F})$ , et définit par  $\rho = [x_i \mapsto t_i]_{1 \leq i \leq n}$ .

Posons  $\mathcal{R}en(l) = (l', E)$ . Remarquons que si  $t$  est une instance de  $l$  alors  $t$  est aussi une instance de  $l'$ , posons  $t = l'\rho'$  avec  $\rho' : \mathcal{Y} \mapsto \mathcal{T}(\mathcal{F})$ , et définit par  $\rho' = [y_i \mapsto t_i]_{1 \leq i \leq n}$ . Comme  $l'$  est linéaire,  $\rho'$  permet de définir une  $\mathcal{Q}$ -substitution  $\sigma'$  par  $\sigma'(y_i) = q_i$ <sup>7</sup> si  $\rho'(y_i) = t_i$  et  $t_i \rightarrow_{\Delta}^* q_i$  pour tout  $1 \leq i \leq n$  et  $t \rightarrow_{\Delta}^* l'\sigma'$  et  $l'\sigma' \rightarrow_{\Delta}^* q$ .

A partir de  $\sigma'$ , nous définissons la  $\mathcal{Q}$ -substitution  $\sigma$  de la manière suivante :  $\sigma(x) = q_j$  si  $\sigma'(x) = q_j$  ou s'il existe une contrainte  $x = y$  de  $E$  telle que  $\sigma'(x) = q_j$ . Comme  $\mathcal{A}$  respecte la condition de linéarité,  $\sigma$  est bien définie et nous avons  $t \rightarrow_{\Delta}^* l\sigma$  et  $l\sigma \rightarrow_{\Delta}^* q$ . ◁

**Automate clos pour un système de réécriture non linéaire**

**THEOREME 2: Automate clos par  $\mathcal{R}$**

Pour tout automate d'arbre  $\mathcal{A}$ , pour tout système de réécriture  $\mathcal{R}$ , pour toute abstraction  $\alpha$  de  $\mathcal{T}(\mathcal{F}, \mathcal{Q})$ , si l'automate  $\mathcal{A}_{\mathcal{R}, \alpha}^*$  existe et vérifie la condition de linéarité induite par  $\mathcal{R}$  alors  $\mathcal{A}_{\mathcal{R}, \alpha}^*$  est clos par  $\rightarrow_{\mathcal{R}^*}$ .

**Démonstration:** ▷

Soit  $\mathcal{R}$  un système de réécriture

$\alpha$  une fonction d'abstraction de  $\mathcal{T}(\mathcal{F}, \mathcal{Q})$

$\mathcal{A} = \langle \mathcal{F}, \mathcal{Q}, \mathcal{Q}_f, \Delta \rangle$  un automate

$q \in \mathcal{Q}$

$\mathcal{A}_{\mathcal{R}, \alpha}^* = \langle \mathcal{F}, \mathcal{Q}, \mathcal{Q}_f, \Delta' \rangle$  automate limite obtenu par complétion de  $\mathcal{A}$  pour  $\mathcal{R}$  et suivant  $\alpha$

$l \rightarrow r$  une règle de réécriture,

$t$  un terme reconnu par  $q$  dans  $\mathcal{A}_{\mathcal{R}, \alpha}^*$  tel que  $t \rightarrow_{\mathcal{R}} t'$  par application de la règle  $l \rightarrow r$ .

---

<sup>7</sup>Les états  $q_i$  existent pour  $1 \leq i \leq n$  car  $t$  est reconnu par  $\mathcal{A}$

Nous pouvons toujours supposer que  $l \rightarrow r$  est appliquée au sommet de  $t$ <sup>8</sup>.

Comme  $\mathcal{A}_{\mathcal{R},\alpha}^*$  respecte la condition de linéarité, il existe une  $\mathcal{Q}$ -substitution  $\sigma$  telle que  $t \rightarrow_{\Delta}^* l\sigma$  et  $\sigma \rightarrow_{\Delta}^* q$ . Par définition de  $\mathcal{A}_{\mathcal{R},\alpha}^*$   $Norm_{\alpha}(r\sigma \rightarrow q) \subset \Delta'$  et donc  $r\sigma \rightarrow_{\Delta'}^* q$ , comme  $t' \rightarrow_{\Delta'}^* r\sigma$  nécessairement  $t' \rightarrow_{\Delta'}^* q$  donc  $t'$  est reconnu par  $\mathcal{A}_{\mathcal{R},\alpha}^*$ .

◁

**Corollaire: 1** *Le langage reconnu par  $\mathcal{A}_{\mathcal{R},\alpha}^*$  contient  $\mathcal{R}^*(\mathcal{L}(\mathcal{A}))$ .*

**Démonstration:** ▷

Puisqu'il contient  $\mathcal{L}(\mathcal{A})$  et qu'il est clos par  $\rightarrow_{\mathcal{R}}^*$ . ◁

### Complexité de la vérification de la condition de linéarité

L'automate intersection est construit en  $\mathcal{O}(\|\mathcal{A}_1\| \cdot \|\mathcal{A}_2\|)$ . La décision du vide d'un automate est un problème *PTIME Complet* [Jac96]. Cette manière de vérifier la condition de linéarité est donc dans *PTIME* pour chaque substitution vérifiant les conditions 1. et 2. du théorème 1.

#### 3.2.2 Condition pour que $\mathcal{A}_{\mathcal{R},\alpha}^*$ existe toujours

Nous allons montrer que si la fonction d'abstraction est totale et n'utilise qu'un nombre fini de nouveaux états alors le processus de complétion termine. Cela revient à démontrer qu'il existe un nombre maximal de transitions possibles dans l'automate, ce qui implique que le processus de complétion ne peut pas créer indéfiniment de nouvelles transitions et que donc la complétion termine toujours.

Considérons une fonction d'abstraction totale  $\alpha$  n'utilisant qu'un nombre fini d'états, un automate  $\mathcal{A} = \langle \mathcal{F}, \mathcal{Q}, \mathcal{Q}_f, \Delta \rangle$ , et posons  $s$  le nombre de symboles apparaissant dans  $\mathcal{F}$ ,  $a$  l'arité maximale d'un symbole de  $\mathcal{F}$  et  $q$  le nombre total d'états comprenant  $\mathcal{Q}$  et les états créés par  $\alpha$ . Un rapide calcul de dénombrement montre qu'au pire on peut former  $s \times q^a \times q$  transitions normalisées. Le nombre de transitions d'un automate  $\mathcal{A}_{\mathcal{R},\alpha}^n$  est borné, par définition le nombre de transition de  $\mathcal{A}_{\mathcal{R},\alpha}^n$  augmente avec  $n$ , donc  $\mathcal{A}_{\mathcal{R},\alpha}^* = \lim_{n \rightarrow \infty} \mathcal{A}_{\mathcal{R},\alpha}^n$  existe et est fini.

Malheureusement si le fait de n'utiliser qu'un nombre fini d'états permet de terminer le processus de complétion, cela peut être au prix d'une sur-approximation :

Considérons l'automate

<b>Automaton Nat</b>	
<b>States</b>	
$qx \quad qy \quad qf \quad qg$	
<b>Final States</b>	
$qf$	
<b>Transitions</b>	
$0$	$\rightarrow \quad q0$
$Succ(q0)$	$\rightarrow \quad qf$

<sup>8</sup>sinon il existe un contexte  $C$  tel que  $t = C[l\sigma] \ (\exists q'l\sigma \rightarrow_{\Delta}^* q')$  et  $C[q'] \rightarrow_{\Delta'}^* q$ ,  $r\sigma \rightarrow_{\Delta'}^* q'$  donc  $C[r\sigma] \rightarrow_{\Delta'}^* q$

et le système de réécriture réduit à la règle

$$\text{Succ}(x) \rightarrow \text{Succ}(\text{Succ}(\text{Succ}(x)))$$

La complétion produit une transition non normalisée :  $\text{Succ}(\text{Succ}(\text{Succ}(q_0))) \rightarrow q_f$ , si on utilise une fonction d'abstraction qui n'utilise que l'état  $q_0$  alors  $\text{Norm}_\alpha(\text{Succ}(\text{Succ}(\text{Succ}(q_0))) \rightarrow q_f) = \{\text{Succ}(q_0) \rightarrow q_0, \text{Succ}(q_0) \rightarrow q_f\}$ . La complétion s'achève car l'étape suivante ne produit pas de nouvelles transitions.

L'automate initial reconnaissait  $1 = \text{Succ}(0)$ , la règle de réécriture  $\text{Succ}(x) \rightarrow \text{Succ}(\text{Succ}(\text{Succ}(x)))$  transforme  $x + 1$  en  $x + 3$ , l'automate  $\mathcal{A}_{\mathcal{R}, \alpha}^*$  ne devrait reconnaître que les nombres impairs, mais  $2 = \text{Succ}(\text{Succ}(0))$  est reconnu par l'automate puisque  $\text{Succ}(\text{Succ}(0)) \rightarrow_{\Delta} \text{Succ}(q_0) \rightarrow_{\Delta} q_f$ . La complétion est réussie mais le langage obtenu est un sur-ensemble de  $\mathcal{R}^*(\mathcal{L}(\mathcal{A}))$ . Nous étudierons en 3.4 des conditions sur l'abstraction pour que  $\mathcal{L}(\mathcal{R}^*(\mathcal{A}))$  soit exactement  $\mathcal{R}^*(\mathcal{L}(\mathcal{A}))$ . Avant cela, nous proposons un algorithme de filtrage dans les automates permettant d'améliorer la recherche de paires critiques.

### 3.3 Filtrage dans un automate

Pour une règle de réécriture  $l \rightarrow r$  et un automate  $\mathcal{A} = \langle \mathcal{F}, \mathcal{Q}, \mathcal{Q}_f, \Delta \rangle$ , nous cherchons ici l'ensemble des  $\mathcal{Q}$ -substitutions  $\sigma$  telles qu'il existe un état  $q \in \mathcal{Q}$  et  $l\sigma \rightarrow_{\Delta}^* q$ .

#### 3.3.1 Algorithmes existants

##### Algorithme naïf

Une solution naïve serait d'énumérer l'ensemble des  $\mathcal{Q}$ -substitutions  $\sigma$  possibles puis vérifier qu'il existe  $q$  tel que  $l\sigma \rightarrow_{\Delta}^* q$ . En pratique si  $x$  est le nombre de variables de  $l$  et  $Q$  le cardinal de  $\mathcal{Q}$ , il y aurait  $Q^x$  substitutions à examiner.

##### Algorithme proposé dans [Gen98]

Cet algorithme est proche d'un algorithme de filtrage standard, il procède par application de règles de déduction sur des formules particulières nommées *problèmes de filtrage*, une  $\mathcal{Q}$ -substitution est une solution d'un problème de filtrage.

**Définition 31 (Problèmes de filtrage)** *un problème de filtrage atomique est soit  $\perp$  soit  $s \trianglelefteq c$  où  $s \in \mathcal{T}(\mathcal{F}, \mathcal{X})$  et  $c \in \mathcal{T}(\mathcal{F}, \mathcal{Q})$ . Un problème de filtrage se définit par induction :*

- Les problèmes de filtrage atomiques sont des problèmes de filtrage.
- si  $\Phi_1$  et  $\Phi_2$  sont des problèmes de filtrage alors :
  - $\Phi_1 \vee \Phi_2, \Phi_1 \wedge \Phi_2$  sont encore des problèmes de filtrage.

**Définition 32 (Solution de problème de filtrage)** *Soit  $\Phi$  et  $\Psi$  deux problèmes de filtrage,  $s \in \mathcal{T}(\mathcal{F}, \mathcal{X})$  et  $c \in \mathcal{T}(\mathcal{F}, \mathcal{Q})$  et  $\mathcal{A} = \langle \mathcal{F}, \mathcal{Q}, \mathcal{Q}_f, \Delta \rangle$  un automate d'arbre, une  $\mathcal{Q}$ -substitution  $\sigma$  est une solution de*

- $s \trianglelefteq c$  si  $s\sigma \rightarrow_{\Delta}^* c$
- $\Phi \vee \Psi$  si  $\sigma$  est une solution de  $\Phi$  ou  $\Psi$
- $\Phi \wedge \Psi$  si  $\sigma$  est une solution de  $\Phi$  et  $\Psi$

Pour une règle de réécriture  $l \rightarrow r$ , chercher l'ensemble des  $\mathcal{Q}$ -substitutions  $\sigma$  telles qu'il existe un état  $q \in \mathcal{Q}$  et  $l\sigma \rightarrow_{\Delta}^* q$ , consiste à trouver toutes les solutions des problèmes de filtrage  $l \trianglelefteq q$  pour  $q \in \mathcal{Q}$  à l'aide des règles de normalisation proposée en figure 3.2.

1. <b>Clash</b>	$\frac{f(s_1, \dots, s_n) \trianglelefteq g(q_1, \dots, q_n)}{\perp}$
si ( $f \neq g$ )	
2. <b>Decompose</b>	$\frac{f(s_1, \dots, s_n) \trianglelefteq f(q_1, \dots, q_n)}{s_1 \trianglelefteq q_1 \wedge \dots \wedge s_n \trianglelefteq q_n}$
3. <b>Configuration</b>	$\frac{s \trianglelefteq q}{\bigvee_{c \rightarrow \Delta q} s \trianglelefteq c \vee \perp}$
4. <b>Or</b>	$\frac{\Phi \vee \perp}{\Phi}$
5. <b>And</b>	$\frac{\Phi \wedge \perp}{\perp}$
6. <b>Distrib</b>	$\frac{\Phi \vee (\Psi_1 \wedge \Psi_2)}{(\Phi \wedge \Psi_1) \vee (\Phi \wedge \Psi_2)}$

FIG. 3.2 – Les règles de déduction permettant de normaliser un problème de filtrage

### 3.3.2 Algorithme proposé

Nous proposons ici un algorithme permettant de représenter un système de réécriture  $\mathcal{R}$  par un automate  $\mathcal{A}_{\mathcal{R}}$ , nous montrerons ensuite comment il est possible de calculer les paires critiques entre  $\mathcal{R}$  et un langage reconnu par un automate  $\mathcal{A}$  grâce à  $\mathcal{A} \cap \mathcal{A}_{\mathcal{R}}$ . L'idée repose sur la représentation sous forme d'arbre d'un automate et d'un système de réécriture, cf. figure 3.3. Nous expliquons tout d'abord comment construire un automate reconnaissant exactement les membres gauches de règles d'un système de réécriture.

#### Instance d'un terme dans un langage

Pour tout terme  $t \in \mathcal{T}(\mathcal{F}, \mathcal{X})$ , nous définissons un automate  $\mathcal{A}_t$  tel que  $\mathcal{L}(\mathcal{A}_t) = \{t\}$ . Pour cela nous utilisons la fonction de normalisation des transitions décrit par la définition 25.

**Définition 33 (Automate d'un terme)** Soient  $t \in \mathcal{T}(\mathcal{F}, \mathcal{X})$ ,  $\mathcal{S}$  l'ensemble des sous termes de  $t$ ,  $\mathcal{Q}$  un ensemble d'états,  $\alpha : \mathcal{S} \rightarrow \mathcal{Q}$  une abstraction de  $\mathcal{S}$  injective et totale, nous définissons  $\mathcal{A}_{\alpha,t}$  automate associé à  $t$ , ou automate du terme  $t$  :

$$\mathcal{A}_{\alpha,t} = \langle \mathcal{F}, \{\alpha(t_i) | t_i \in \mathcal{S}\}, \{\alpha(t)\}, \text{Norm}_{\alpha}(t \rightarrow \alpha(t)) \rangle$$

Dans le souci d'alléger les notations, et considérant que la fonction d'abstraction  $\alpha$  n'a qu'une importance mineure du moment que c'est une abstraction injective, nous noterons plus souvent  $\mathcal{A}_{\alpha,t}$  par  $\mathcal{A}_t$ .

**Exemple 11** Considérons le terme  $\text{Plus}(x, \text{Succ}(\text{Plus}(y, \text{Succ}(0))))$  et la fonction  $\alpha$  définie par :

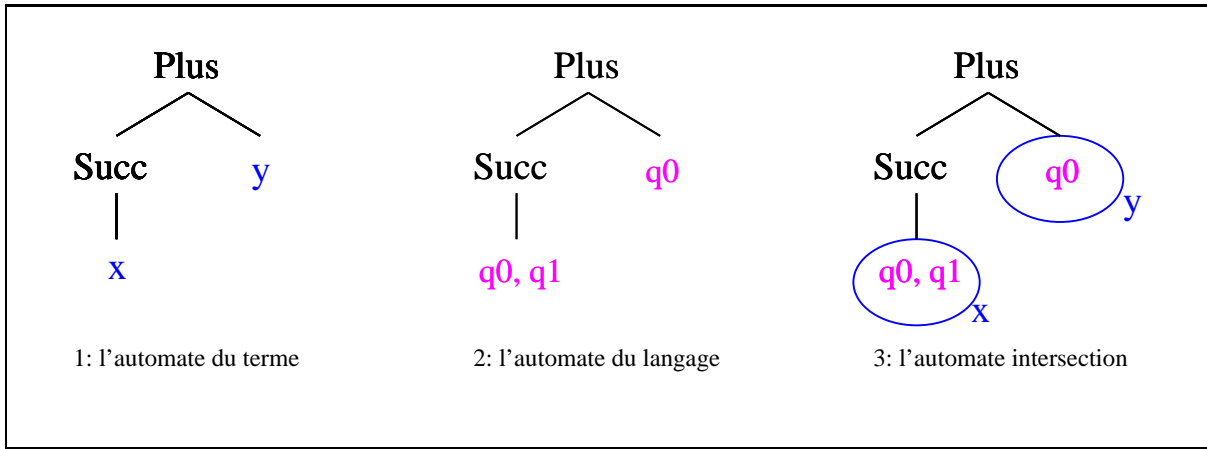


FIG. 3.3 – superposition d'arbres

- $\alpha(0) = q_0$
- $\alpha(\text{Succ}(0)) = q_1$
- $\alpha(y) = q_y$
- $\alpha(x) = q_x$
- $\alpha(\text{Plus}(y, \text{Succ}(0))) = q_2$
- $\alpha(\text{Succ}(\text{Plus}(y, \text{Succ}(0)))) = q_3$
- $\alpha(\text{Plus}(x, \text{Succ}(\text{Plus}(y, \text{Succ}(0))))) = q_t$

D'après la définition précédente, l'automate du terme  $\text{Plus}(x, \text{Succ}(\text{Plus}(y, \text{Succ}(0))))$  est

$$\mathcal{Q} = \{q_0, q_1, q_x, q_y, q_2, q_3, q_t\}$$

$$\mathcal{Q}_f = \{q_t\}$$

$$\Delta = \{ 0 \rightarrow q_0 \\ \text{Succ}(q_0) \rightarrow q_1 \\ y \rightarrow q_y \\ \text{Plus}(q_y, q_1) \rightarrow q_2 \\ \text{Succ}(q_2) \rightarrow q_3 \\ x \rightarrow q_x \\ \text{Plus}(q_x, q_3) \rightarrow q_t \}$$

**Proposition 4** Soit  $t \in \mathcal{T}(\mathcal{F}, \mathcal{X})$  et  $\mathcal{A}_t$  son automate,

$$\mathcal{L}(\mathcal{A}_t) = \{t\}$$

**Démonstration:**  $\triangleright$

Nous raisonnons par récurrence sur la profondeur de  $t$  :

- Si  $t$  est de profondeur 1 alors  $\mathcal{A}_t = \langle \mathcal{F}, \{\alpha(t)\}, \{\alpha(t)\}, \{t \rightarrow \alpha(t)\} \rangle$ , donc  $\mathcal{L}(\mathcal{A}_t) = \{t\}$
- Supposons que si  $t$  est de profondeur inférieure ou égale à  $n$  alors  $\mathcal{L}(\mathcal{A}_t) = \{t\}$
- Soient  $t$  de profondeur  $n + 1$ ,  $\mathcal{A}_t = \langle \mathcal{F}, \bigcup_{t_i \in \mathcal{S}} \{\alpha(t_i)\}, \{\alpha(t)\}, \text{Norm}_\alpha(t \rightarrow \alpha(t)) \rangle$ , posons  $t = f(t_1, \dots, t_m)$  par hypothèse de récurrence  $t_i$  est reconnu par  $\alpha(t_i)$  dans  $\mathcal{A}_t$  pour  $1 \leq i \leq m$  donc  $t = f(t_1, \dots, t_m) \rightarrow_{\Delta}^* f(\alpha(t_1), \dots, \alpha(t_m)) \rightarrow_{\Delta} \alpha(t)$ , donc  $\mathcal{L}(\mathcal{A}_t) = \{t\}$ .

$\triangleleft$

**Définition 34 (Automate des instances)** Soit  $\mathcal{A} = \langle \mathcal{F}, \mathcal{Q}, \mathcal{Q}_f, \Delta \rangle$  un automate et  $\mathcal{A}_t$  l'automate d'un terme  $t \in \mathcal{T}(\mathcal{F}, \mathcal{X})$  suivant une abstraction  $\alpha$ , nous définissons  $\mathcal{A}_{\mathfrak{m}} = \langle \mathcal{F}, \mathcal{Q}_{\mathfrak{m}}, \mathcal{Q}_{f,\mathfrak{m}}, \Delta_{\mathfrak{m}} \rangle$  l'automate des instances de  $t$  dans  $\mathcal{A}$  par :

$$\begin{aligned} \mathcal{Q}_{\mathfrak{m}} &= \mathcal{Q}_t \times \mathcal{Q} \\ \mathcal{Q}_{f,\mathfrak{m}} &= \{\alpha(t)\} \times \mathcal{Q} \\ \Delta_{\mathfrak{m}} &= \{(x, q) \rightarrow (q_x, q) \mid x \in \mathcal{X}, x \rightarrow q_x \in \Delta_t, q \in \mathcal{Q}\} \cup \\ &\quad \{f((q_1, q'_1), \dots, (q_n, q'_n)) \rightarrow (q_{n+1}, q'_{n+1}) \mid \begin{array}{l} q_i \in \mathcal{Q}_t, q'_i \in \mathcal{Q} \\ f(q_1, \dots, q_n) \rightarrow q_{n+1} \text{ dans } \mathcal{A}_t \\ f(q'_1, \dots, q'_n) \rightarrow q'_{n+1} \text{ dans } \mathcal{A} \end{array}\} \end{aligned}$$

**Définition 35 (Q-instances)** Soit  $\mathcal{F}$  un alphabet,  $\mathcal{Q}$  un ensemble d'états,  $\mathcal{X}$  un ensemble de variables, nous définissons l'ensemble des  $\mathcal{Q}$ -instances par induction :

- tous les couples  $(x, q)$  pour  $x \in \mathcal{X}$  et  $q \in \mathcal{Q}$  sont des  $\mathcal{Q}$ -instances
- pour tout symbole de fonction  $f$  d'arité  $n$ , si  $i_1, \dots, i_n$  sont des  $\mathcal{Q}$ -instances alors  $f(i_1, \dots, i_n)$  est encore une  $\mathcal{Q}$ -instance.

**Définition 36 (Langage de l'automate des instances d'un terme dans un langage)** Soit  $\mathcal{A} = \langle \mathcal{F}, \mathcal{Q}, \mathcal{Q}_f, \Delta \rangle$  un automate,  $t$  un terme représenté par l'automate  $\mathcal{A}_t$  et  $\mathcal{A}_{\mathfrak{m}}$  l'automate des instances de  $t$  dans  $\mathcal{A}$ . L'automate  $\mathcal{A}_{\mathfrak{m}}$  reconnaît l'ensemble des  $\mathcal{Q}$ -instances  $i$  telles qu'il existe un état final  $q$  de  $\mathcal{A}_{\mathfrak{m}}$  vérifiant  $i \rightarrow_{\Delta_{\mathfrak{m}}}^* q$

**Définition 37 (Q-substitutions définies par une Q-instance)** Soient  $\mathcal{A} = \langle \mathcal{F}, \mathcal{Q}, \mathcal{Q}_f, \Delta \rangle$  un automate,  $t$  un terme,  $\mathcal{A}_t$  son automate et  $\mathcal{A}_{\mathfrak{m}}$  l'automate des instances de  $t$  dans  $\mathcal{A}$ . Soit  $s$  une  $\mathcal{Q}$ -instance si  $s$  est reconnu  $\mathcal{A}_{\mathfrak{m}}$  alors  $s$  définit par induction un ensemble de  $\mathcal{Q}$ -substitutions  $\sigma$

- si  $s = (x, q)$ , alors  $s$  définit la  $\mathcal{Q}$ -substitution  $[x \leftarrow q]$
- si  $s = (a, a)$  alors  $s$  définit la substitution de domaine vide
- si  $s = f(s_1, \dots, s_n)$ , posons  $\{\sigma_{i,j}\}_{i \in I_j}$  les substitutions définies par  $s_j$ ,  $s$  définit l'ensemble des substitutions  $\tau$  obtenues par combinaison  $\tau = \sigma_{1,i_1} \circ \dots \circ \sigma_{n,i_n}$ .

**Exemple 12** Considérons l'automate du terme  $f(x, g(y))$  défini par :

<b>Automaton</b>	<i>Terme</i>
<b>States</b>	
$qx \quad qy \quad qf \quad qg$	
<b>Final States</b>	
$qf$	
<b>Transitions</b>	
$y$	$\rightarrow \quad qy$
$g(qy)$	$\rightarrow \quad qg$
$x$	$\rightarrow \quad qx$
$f(qx, qg)$	$\rightarrow \quad qf$

lors de l'exemple 11 et l'automate reconnaissant le langage  $\{f(a, g^+(b)), f(g^+(b), a)\}$

**Automaton** *Langage*

**States**

$q1\ q2\ q3\ q4$

**Final States**

$q4$

**Transitions**

$a \rightarrow q1$

$b \rightarrow q2$

$g(q2) \rightarrow q3$

$g(q3) \rightarrow q3$

$f(q1, q3) \rightarrow q4$

$f(q3, q1) \rightarrow q4$

*l'automate des instances de  $f(x, g(y))$  dans  $\{f(a, g^+(b)), f(g^+(b), a)\}$  :*

**Automaton** *Instances*

**States**

$\{qx, qy, qf, qg\} \times \{q1, q2, q3, q4\}$

**Final States**

$\{qf\} \times \{q1, q2, q3, q4\}$

**Transitions**

$(x, q1) \rightarrow (qx, q1)$

$(x, q2) \rightarrow (qx, q2)$

$(x, q3) \rightarrow (qx, q3)$

$(x, q4) \rightarrow (qx, q4)$

$(y, q1) \rightarrow (qy, q1)$

$(y, q2) \rightarrow (qy, q2)$

$(y, q3) \rightarrow (qy, q3)$

$(y, q4) \rightarrow (qy, q4)$

$g((qy, q2)) \rightarrow (qg, q2)$

$g((qy, q3)) \rightarrow (qg, q3)$

$f((qx, q1), (qg, q3)) \rightarrow (qf, q4)$

$f((qx, q3), (qg, q1)) \rightarrow (qf, q4)$

*Cet automate ne reconnaît qu'une seule  $\mathcal{Q}$ -instance :  $f((x, q1), g((y, q3)))$  qui définit la  $\mathcal{Q}$ -substitution  $\sigma = [x \leftarrow q1, y \leftarrow q3]$*

**THEOREME 3:**

Soient  $\mathcal{A} = \langle \mathcal{F}, \mathcal{Q}, \mathcal{Q}_f, \Delta \rangle$  un automate,  $l \rightarrow r$  une règle de réécriture, et  $\mathcal{A}_{\cap}$  l'automate des instances de  $l$  dans  $\mathcal{A}$ , si  $\mathcal{A}$  respecte la condition de linéarité induite par  $l \rightarrow r$  alors l'ensemble des  $\mathcal{Q}$ -substitutions défini par les états  $(\alpha(l), q)$  de  $\mathcal{A}_{\cap}$  est exactement l'ensemble des  $\mathcal{Q}$ -substitutions  $\{\sigma_i\}_{i \in I}$  telles que  $l\sigma_i \rightarrow_{\Delta}^* q$  dans  $\mathcal{A}$ .

**Démonstration:**  $\triangleright$

Montrons la double inclusion :

1. Supposons tout d'abord que  $\{\sigma_i\}$  est un ensemble de  $\mathcal{Q}$ -substitutions définies par un terme  $s$  tel que  $s \rightarrow^*(\alpha(l), q)$  dans  $\mathcal{A}_{\cap}$  montrons qu'alors  $l\sigma \rightarrow^* q$ , par récurrence sur la longueur de la dérivation  $s \rightarrow^n(\alpha(l), q)$  :

- si  $s \rightarrow(\alpha(l), q)$  alors nécessairement  $s$  est de profondeur 1 et donc d'une des formes suivantes :
  - $(a, a)$  où  $a$  est une constante et par construction de  $\mathcal{A}_{\cap}$ ,  $(a, a) \rightarrow_{\mathcal{A}_{\cap}}(\alpha(a), q)$  donc  $l = a$  et  $l \rightarrow_{\Delta} q$ ,  $s$  définit la substitution de domaine vide.
  - $(x, a)$  où  $x$  est une variable et  $a$  une constante. De même, par construction de  $\mathcal{A}_{\cap}$ ,  $(x, a) \rightarrow_{\mathcal{A}_{\cap}}(\alpha(a), q)$  donc  $l = x$ ,  $s$  définit la  $\mathcal{Q}$ -substitution  $\sigma = [x \leftarrow q]$ , nous avons  $l\sigma = a$  et  $a \rightarrow q$ .

- Posons comme hypothèse de récurrence que si un ensemble  $\{\sigma_i\}_{i \in I}$  défini par un terme  $s$  de  $\mathcal{A}_{\cap}$  vérifiant  $s \rightarrow^k(\alpha(l), q)$  pour tout  $k \leq n$  alors  $l\sigma_i \rightarrow^* q$  dans  $\mathcal{A}$ .

- Considérons maintenant un ensemble de substitutions  $\{\tau_i\}_{i \in I}$  défini par un terme  $s$  tel que  $s \rightarrow^m(\alpha(l), q)$ . Nécessairement  $s$  est un terme de la forme  $f(s_1, \dots, s_n)$ , avec  $s_j = (l_j, q_j)$  et détermine, par définition, un ensemble de  $\mathcal{Q}$ -substitutions  $\{\tau_i\}_{i \in I}$  obtenu par union  $\tau_i = \sigma_{1, i_1} \circ \dots \circ \sigma_{n, i_n}$ , où  $\sigma_{j, i_j}$  est l'ensemble des  $\mathcal{Q}$ -substitutions définies récursivement par  $s_j$ .

$f(s_1, \dots, s_n) \rightarrow_{\mathcal{A}_{\cap}}(\alpha(l), q)$  donc  $l$  est de la forme  $f(l_1, \dots, l_n)$  et il existe des états  $q_j$  tels que  $s_j \rightarrow_{\mathcal{A}_{\cap}}(\alpha(l_j), q_j)$ . Nous pouvons appliquer l'hypothèse de récurrence :  $s_j \rightarrow^k(\alpha(l_j), q_j)$  avec  $k \leq n$ , donc  $l_j\sigma_j \rightarrow^* q_j$  dans  $\mathcal{A}$  et pour tout  $\tau_i$  il existe une combinaison des  $\sigma_{i, j}$  vérifiant  $l\tau_i \rightarrow f(l_1\sigma_{1, i_1}, \dots, l_n\sigma_{n, i_n}) \rightarrow f(q_1, \dots, q_n) \rightarrow_{\Delta}^* q$ .

2. Montrons maintenant, par induction structurale sur  $l$ , que si  $\sigma$  est une  $\mathcal{Q}$ -substitution telle que  $l\sigma \rightarrow_{\Delta}^* q$ , alors il existe un terme  $s$  reconnu par  $\mathcal{A}_{\cap}$  tel que  $\sigma$  soit une substitution définie par  $s$  :

(a) si  $l$  est une constante  $a$

Montrons qu'il existe un terme de  $\mathcal{A}_{\cap}$  qui définit la substitution de domaine vide : Nous savons que d'une part  $a \rightarrow_{\mathcal{A}_t} \alpha(a)$  et que d'autre part, si  $a$  est instanciable dans  $\mathcal{A}$  il existe un état  $q$  de  $\mathcal{A}$  tel que  $a \rightarrow_{\mathcal{A}} q$  donc par construction de  $\mathcal{A}_{\cap}$   $a \rightarrow_{\mathcal{A}_{\cap}}(\alpha(a), q)$  comme  $(\alpha(a), q)$  est un état final de  $\mathcal{A}_{\cap}$ ,  $\mathcal{A}_{\cap}$  reconnaît  $a$  et  $a$  définit bien la substitution de domaine vide.

(b) si  $l$  est une variable  $x$

D'une part  $x \rightarrow_{\mathcal{A}_t} \alpha(x)$  et d'autre part l'ensemble des instances de  $l$  dans  $\mathcal{A}$  est l'ensemble des termes reconnus par un état de  $\mathcal{A}$ . Par construction  $\{(x, q) \rightarrow (\alpha(x), q) | q \in$

$\mathcal{Q}\} \subset \Delta_{\bar{m}}$  et comme  $\mathcal{Q}_{f,\bar{m}} = \{\alpha(x)\} \times \mathcal{Q}$  nous avons bien défini la famille de  $\mathcal{Q}$ -substitution de domaine réduit à un élément  $[x \leftarrow q]$ .

(c) si  $l$  est de la forme  $f(l_1, \dots, l_n)$

$\mathcal{A}$  reconnaît  $l\sigma$  donc il existe des états  $q, q_1, \dots, q_n$  de  $\mathcal{A}$  tels que  $f(q_1, \dots, q_n) \rightarrow_{\mathcal{A}}^* q$ ,  $l_i\sigma \rightarrow_{\mathcal{A}} q_i$ . Posons  $s = f((\alpha(l_1), q_1), \dots, (\alpha(l_n), q_n))$ , s'il existe états  $q_i$  de  $\mathcal{A}$  tels que  $q_i$  reconnaît  $t_i\sigma$  alors par hypothèse de récurrence il existe des termes  $s_i$  de  $\mathcal{A}_{\bar{m}}$  qui définissent des substitutions  $\sigma_i$  telles que  $\sigma = \sigma_1 \circ \dots \circ \sigma_n$ .

– Soit  $\sigma_1 \circ \dots \circ \sigma_n$  n'est pas défini et  $l\sigma$  n'est pas reconnu par  $\mathcal{A}$ , contradiction.

– Soit  $\sigma_1 \circ \dots \circ \sigma_n$  est bien défini et  $\mathcal{A}_{\bar{m}}$  reconnaît  $s = f(s_1, \dots, s_n)$  et  $s$  définit  $\sigma$ .

◁

L'implémentation de cet algorithme est donnée en annexe A.

### 3.3.3 Comparaison des algorithmes, complexité du problème

Rappelons que pour un automate  $\mathcal{A}$  et un terme  $t$  nous cherchons toutes les  $\mathcal{Q}$ -substitutions  $\sigma$  et tous les états  $q$  tels que  $t\sigma \rightarrow_{\Delta}^* q$ . Un problème plus général est de savoir s'il existe au moins une instance close de  $t$  qui soit acceptée par  $\mathcal{A}$ . Nous pouvons réduire notre problème au *Ground Instance Intersection Problem* : si nous trouvons un ensemble non vide de  $\mathcal{Q}$ -substitutions  $\sigma$  telles que  $t\sigma \rightarrow_{\Delta}^* q$  alors il existe au moins une instance close de  $t$  acceptée par  $\mathcal{A}$ .

Ce second problème est polynomial si  $t$  est linéaire et  $\mathcal{A}$  déterministe, *NP-complet* si  $t$  n'est pas linéaire et  $\mathcal{A}$  déterministe, *EXPTIME-complet* si  $t$  n'est pas linéaire et  $\mathcal{A}$  non déterministe. [CDG<sup>+</sup>97].

La complexité du premier algorithme est exponentielle même si l'automate est déterministe, que le terme soit linéaire ou non, que l'automate soit déterministe ou non. Le premier algorithme revient en effet à mettre une formule sous forme normale disjonctive, la règle **Distrib** combinée à la règle **Configuration** impliquent une complexité exponentielle. Nous n'avons pas su déterminer la complexité de notre algorithme, en revanche, nous avons pu constater qu'en pratique il était environ six fois plus rapide et que la taille occupée par le processus n'explosait pas comme c'était le cas avec l'algorithme précédent. Nous avons constaté que plus l'automate du langage utilisait des états distincts pour normaliser des configurations distinctes plus la recherche était facilitée. Le problème est que le détail de l'automate est directement lié à l'approximation utilisée lors de la complétion. Plus l'approximation est détaillée plus l'automate le sera, mais la complétion risque très fortement de diverger. Tout l'art de la complétion réside dans le choix d'une approximation suffisamment fine pour que le calcul des paires critiques reste *faisable*, mais tout de même suffisante pour terminer la complétion.

## 3.4 Rôle de l'approximation

Nous venons de montrer comment calculer un automate  $\mathcal{A}_{\mathcal{R},\alpha}^*$  représentant un sur-ensemble des termes atteignables à partir d'un langage régulier représenté par un automate  $\mathcal{A}$ , d'un système de réécriture  $\mathcal{R}$  et une abstraction  $\alpha$ . Nous disposons maintenant d'une procédure de semi-décision pour savoir si un ensemble régulier de termes est atteignable ou non, il suffit de représenter cet ensemble par un second automate  $\mathcal{A}'$  et de calculer  $\mathcal{A}_{\mathcal{R},\alpha}^* \cap \mathcal{A}'$ , si cette intersection est vide alors il n'existe aucun terme reconnu par  $\mathcal{A}'$  qui soit aussi un terme atteignable par  $\mathcal{R}$  à partir de  $\mathcal{A}$ . En revanche, si l'intersection n'est pas vide, il n'est pas possible de décider si les

termes reconnus par l'automate intersection sont produits des termes de  $\mathcal{R}^*(\mathcal{L}(\mathcal{A}))$  ou s'ils ont été produits par approximation ou non.

### 3.4.1 Choix de la fonction d'abstraction

Nous proposons ici d'étudier l'importance du choix de la fonction d'approximation en étudiant un exemple de complétion d'un même couple  $(\mathcal{A}, \mathcal{R})$  à l'aide de deux fonctions d'abstraction distinctes.

Considérons l'automate  $\mathcal{A}$  représentant le langage  $\{f(a, g^n(b))\}_{n>0}$  :

<b>Automaton A</b>	
<b>States</b>	
$qa$	$qb$ $q1$ $qf$
<b>Final States</b>	
	$qf$
<b>Transitions</b>	
$a$	$\rightarrow qa$
$b$	$\rightarrow qb$
$g(qb)$	$\rightarrow q1$
$g(q1)$	$\rightarrow q1$
$f(qa, q1)$	$\rightarrow qf$

Nous cherchons l'ensemble des termes atteignables par la règle  $f(x, y) \rightarrow f(y, g(x))$  à partir de  $\mathcal{A}$ . Ensuite nous essayerons de montrer que les termes du langage  $f(g^n(b), a)$  sont inatteignables. La première étape de complétion de  $\mathcal{A}$  produit la transition suivante :

$$f(q1, g(qa)) \rightarrow qf$$

#### 1. une approximation trop forte

Pour commencer nous choisissons l'approximation  $\alpha_1$  qui associe  $g(qa)$  à  $qa$ . Cette approximation est suffisante pour normaliser les transitions construites lors des étapes de complétion suivantes.  $\mathcal{A}_{\mathcal{R}, \alpha_1}^*$  est défini par

**Automaton alpha1**

**States**

$qa\ qb\ q1\ qf$

**Final States**

$qf$

**Transitions**

$a \rightarrow qa$

$b \rightarrow qb$

$g(qb) \rightarrow q1$

$g(q1) \rightarrow q1$

$f(qa, q1) \rightarrow qf$

$g(qa) \rightarrow qa$

$f(q1, qa) \rightarrow qf$

Il suffit maintenant d'écrire un automate  $\mathcal{A}'$  reconnaissant le langage  $f(g^n(b), a)$  et de calculer  $I = \mathcal{A}_{\mathcal{R}, \alpha_1}^* \cap \mathcal{A}'$ .

**Automaton alpha1**

**States**

$qa\ qb\ q1\ qf$

**Final States**

$qf$

**Transitions**

$a \rightarrow qa$

$b \rightarrow qb$

$g(qb) \rightarrow q1$

$g(q1) \rightarrow q1$

$f(q1, qa) \rightarrow qf$

l'automate intersection :

**Automaton I**

**States**

$q3\ q2\ q1\ q0$

**Final States**

$q3$

**Transitions**

$a \rightarrow q2$

$b \rightarrow q0$

$g(q0) \rightarrow q1$

$g(q1) \rightarrow q1$

$f(q1, q2) \rightarrow q3$

cet automate n'est pas vide, il reconnaît le langage  $f(g^+(b), a)$ . L'automate  $\mathcal{A}_{\mathcal{R}, \alpha_1}^*$  reconnaît un *sur-ensemble* de  $\mathcal{R}^*(\mathcal{A})$ , nous ne pouvons pas affirmer que les termes de la forme  $f(g^n(b), a)$  sont inatteignables. Nous allons refaire la complétion de  $\mathcal{A}$  par  $\mathcal{R}$  en utilisant une autre approximation :

## 2. seconde approximation

Considérons maintenant l'approximation  $\alpha_2$  qui utilise un nouvel état  $q_2$  et définie par

$$\alpha_2(g(q_a)) = q_2,$$

$$\alpha_2(g(q_2)) = q_2,$$

l'automate complet  $\mathcal{A}_{\mathcal{R}, \alpha_2}^*$  :

<b>Automaton Complet</b>	
<b>States</b>	
$q_2 \quad qa \quad qb \quad q_1 \quad qf$	
<b>Final States</b>	
$qf$	
<b>Transitions</b>	
$a$	$\rightarrow qa$
$b$	$\rightarrow qb$
$g(qb)$	$\rightarrow q_1$
$g(q_1)$	$\rightarrow q_1$
$f(qa, q_1)$	$\rightarrow qf$
$g(qa)$	$\rightarrow q_2$
$f(q_1, q_2)$	$\rightarrow qf$
$f(q_2, q_1)$	$\rightarrow qf$
$g(q_2)$	$\rightarrow q_2$

et cette fois l'intersection  $I = \mathcal{A}_{\mathcal{R}, \alpha_2}^* \cap \mathcal{A}'$  est vide, les termes du langage  $f(g^n(b), a)$  sont *inatteignables*.

Cet exemple montre donc l'importance du choix de la fonction d'abstraction. Intuitivement une approximation forte va rassembler sous un même état des termes qui n'étaient pas joignables par réécriture. Une approximation est un calcul exact lorsqu'elle normalise les termes à ajouter par des états qui ne reconnaissaient pas d'autres termes.

### 3.4.2 Calcul exact des descendants

Pour un automate  $\mathcal{A}$  et un système de réécriture  $\mathcal{R}$  donnés, nous avons déjà montré que sous réserve que la condition de linéarité soit vérifiée et que le processus de complétion termine,  $\mathcal{R}^*(\mathcal{L}(\mathcal{A})) \subset \mathcal{L}(\mathcal{A}_{\mathcal{R}, \alpha}^*)$ . Nous nous proposons d'étudier maintenant des conditions suffisantes sur  $\mathcal{R}$  et  $\alpha$  pour que  $\mathcal{L}(\mathcal{A}_{\mathcal{R}, \alpha}^*) = \mathcal{R}^*(\mathcal{L}(\mathcal{A}))$ .

Il a déjà été montré que  $\mathcal{R}^*(\mathcal{L}(\mathcal{A}))$  est régulier si

- $\mathcal{R}$  est clos [DT90]
- $\mathcal{R}$  est linéaire à droite et monadique [Sal88]
- $\mathcal{R}$  est linéaire et semi-monadique [CDGV91]
- $\mathcal{R}$  est *décroissant* [Jac96]

- $\mathcal{L}(\mathcal{A})$  est l'ensemble des instances closes d'un terme linéaire [Rét99] et si  $\mathcal{R}$  est monadique et linéaire à droite.

où monadique (resp. semi-monadique) signifie que le membre droit de chaque règle de  $\mathcal{R}$  est

- soit une variable,
- soit un terme  $f(t_1, \dots, t_n)$  où chaque  $t_i$  est
  - soit une variable,
  - soit une constante (resp. soit une variable, soit un terme clos),

et où décroissant signifie que le membre droit de chaque règle de  $\mathcal{R}$  est

- soit une variable,
- soit un terme  $f(t_1, \dots, t_n)$  où chaque  $t_i$  est
  - soit une variable,
  - soit un terme clos,

Nous définissons la fonction d'abstraction  $\alpha_0$  qui associe chaque sous terme  $t$  d'une transition à normaliser à un nouvel état. Nous allons montrer que tout processus de complétion réussi avec  $\alpha_0$  est exact.

**THEOREME 4: Calcul exact des descendants**

Soit  $\mathcal{A} = \langle \mathcal{F}, \mathcal{Q}, \mathcal{Q}_f, \Delta \rangle$  un automate,  $\mathcal{R}$  un système de réécriture, si les hypothèses suivantes sont vérifiées :

1. A chaque étape  $i$  de complétion les états de  $\mathcal{A}_{\mathcal{R}, \alpha_0}^i$  sont à la fois utiles et accessibles
2. la complétion est réussie
3. – si  $\mathcal{R}$  est linéaire
  - ou si  $\mathcal{R}$  n'est pas linéaire à gauche mais que la condition de linéarité est vérifiée

alors  $\mathcal{L}(\mathcal{A}_{\mathcal{R}, \alpha_0}^*) = \mathcal{R}^*(\mathcal{L}(\mathcal{A}))$ .

**Démonstration:** ▷

Nous avons déjà montré que si la complétion termine et si la condition de linéarité est vérifiée alors  $\mathcal{L}(\mathcal{A}_{\mathcal{R}, \alpha_0}^*) \supset \mathcal{R}^*(\mathcal{L}(\mathcal{A}))$ .

Nous allons montrer que si la fonction d'abstraction utilisé est  $\alpha_0$  et que à chaque étape  $i$  de complétion tous les états de  $\mathcal{A}_{\mathcal{R}, \alpha_0}^i$  sont à la fois utiles et accessibles alors le processus de complétion est correct au sens où chaque terme ajouté lors d'une étape de complétion est issu d'un terme reconnu par l'automate à l'étape précédente.

Considérons un terme  $t$  tel que  $t \in \mathcal{L}(\mathcal{A}_{\mathcal{R}, \alpha_0}^{n+1})$ ,  $\mathcal{A}_{\mathcal{R}, \alpha_0}^n \vdash_{\mathcal{R}} \mathcal{A}_{\mathcal{R}, \alpha_0}^{n+1}$ , tel que  $t$  n'était déjà reconnu par  $\mathcal{A}_{\mathcal{R}, \alpha_0}^n$ . Il s'agit de montrer qu'il existe  $t_0 \in \mathcal{L}(\mathcal{A}_{\mathcal{R}, \alpha_0}^n)$  tel que  $t_0 \rightarrow_{\mathcal{R}} t$ . Par construction

$$\mathcal{A}_{\mathcal{R}, \alpha_0}^{n+1} = \langle \mathcal{F}, \mathcal{Q}, \mathcal{Q}_f, \Delta^n \cup \bigcup_{l \rightarrow r \in \mathcal{R}, q \in \mathcal{Q}, \sigma \in \Sigma(\mathcal{Q}, \mathcal{X}), l\sigma \rightarrow_{\Delta^n}^* q} Norm_{\alpha_0}(r\sigma \rightarrow q) \rangle$$

Nous raisonnons par induction structurelle sur  $t$  :

1. si  $t$  est une constante  $a$ , telle que  $a \rightarrow q_a$ , par hypothèse  $t$  est reconnu par  $\mathcal{A}_{\mathcal{R}, \alpha_0}^{n+1}$  donc  $q_a \in \mathcal{Q}_f$ . Si  $a$  n'est pas reconnu par  $\mathcal{A}_{\mathcal{R}, \alpha_0}^n$  alors la transition  $a \rightarrow_{\Delta}^* q_a$  a été ajouté par la normalisation d'une règle produite par la complétion et  $\exists l \rightarrow r \in \mathcal{R}, q \in \mathcal{Q}, \sigma \in \Sigma(\mathcal{Q}, \mathcal{X}), l\sigma \rightarrow_{\Delta^n}^* q$  et  $a \rightarrow q_a$  apparaît dans  $Norm_{\alpha_0}(r\sigma \rightarrow q)$ . D'après définition 25 de  $Norm_{\alpha_0}$ ,  $\alpha_0$  associe  $q_a$  à un sous terme de  $r\sigma$ , comme  $q_a$  et  $q$  sont finaux et que  $\alpha_0$  ne produit pas d'état final, nécessairement  $q_a = q$  et  $a = r\sigma$ . Nous avons donc  $l\sigma \rightarrow_{\mathcal{R}} a$ , par ailleurs comme tous les

états de  $\mathcal{A}_{\mathcal{R},\alpha_0}^n$  sont utiles et accessibles, il existe un terme  $t_0$  tel que  $t_0 \rightarrow_{\Delta}^* l\sigma$ , et nous avons  $t_0 \rightarrow t$ .

2. si  $t$  est de la forme  $f(t_1, \dots, t_n)$ , supposons que  $t$  soit reconnu par l'état final  $q_t$  dans  $\mathcal{A}_{\mathcal{R},\alpha_0}^{n+1}$  et que  $t$  ne soit pas reconnu par  $\mathcal{A}_{\mathcal{R},\alpha_0}^n$ .

Nécessairement il existe au moins une règle  $l \rightarrow r$ , une substitution  $\sigma$  et un état  $q$  tels que  $q_t$  apparaisse dans  $Norm_{\alpha_0}(r\sigma \rightarrow q)$ . Comme  $q_t \in Q_f$  et que  $\alpha_0$  n'utilise que des nouveaux états qui ne sont pas des états finaux pour normaliser  $r\sigma$ , clairement  $q$  est final et  $q = q_t$ . Si  $f(t_1, \dots, t_n) \rightarrow_{\Delta}^* q_t$  alors il existe des états  $q_1, \dots, q_n$  de  $\mathcal{A}_{\mathcal{R},\alpha_0}^{n+1}$  tels que  $t_i \rightarrow_{\Delta}^* q_i$ . Si  $t$  n'est pas reconnu par  $\mathcal{A}_{\mathcal{R},\alpha_0}^n$  comme par hypothèse tous les états de  $\mathcal{A}_{\mathcal{R},\alpha_0}^n$  sont utiles et accessibles,  $q_1, \dots, q_n$  n'apparaissent pas dans  $\mathcal{A}_{\mathcal{R},\alpha_0}^n$ .

En revanche la transition  $f(q_1, \dots, q_n)$  apparaît dans  $\mathcal{A}_{\mathcal{R},\alpha_0}^{n+1}$ , ce qui implique que cette transition a été produite par la normalisation de  $r\sigma \rightarrow q_f$ , nous avons  $r\sigma \rightarrow_{\Delta}^* q_f$  et  $f(q_1, \dots, q_n) \rightarrow_{\Delta}^* q_t$ , comme  $\alpha_0$  est bijective et n'utilise que des nouveaux états, nécessairement  $t \rightarrow_{\Delta}^* r\sigma$ .

Il s'agit de montrer que  $t$  n'est pas un terme obtenu par sur-approximation et donc qu'il existe une  $\mathcal{T}(\mathcal{F})$ -substitution  $\tau$  vérifiant  $t = r\sigma\tau$ . Posons  $\sigma = [x_1 \mapsto q'_1, \dots, x_n \mapsto q'_n]$ .

- Si  $\mathcal{R}$  est linéaire à droite, les variables  $x_i$  n'ont qu'une occurrence dans  $\sigma$ , comme par hypothèse tous les états sont utiles et accessibles, il existe  $n$  termes  $t'_1, \dots, t'_n$ , vérifiant  $t'_i \rightarrow_{\Delta}^* q'_i$ , nous définissons alors  $\tau = [x_1 \mapsto t'_1, \dots, x_n \mapsto t'_n]$ .
- Si  $\mathcal{R}$  n'est pas linéaire à droite : si il n'existait aucune substitution  $\tau$  telle que  $t = r\sigma\tau$ , nécessairement il existerait un état  $q'_i$  et deux termes distincts  $t'_1$  et  $t'_2$  tels que les transitions  $t'_1 \rightarrow_{\Delta}^* q'_i$  et  $t'_2 \rightarrow_{\Delta}^* q'_i$  apparaissent déjà dans  $\mathcal{A}_{\mathcal{R},\alpha_0}^n$ <sup>9</sup>, or toute variable non linéaire dans un membre droit de règle est non linéaire dans le membre gauche de règle correspondant, comme  $\mathcal{R}$  respecte la condition de linéarité (gauche), nous aboutissons à une contradiction.

Pour finir, comme précédemment, puisque tous les états sont utiles et accessibles, il existe au moins un terme  $t_0 \in \mathcal{T}(\mathcal{F})$  tel que  $t_0 \rightarrow_{\Delta}^* l\sigma$ .

◁

### 3.5 Utiliser la complétion d'automates pour des preuves de propriétés

Supposons que nous étudions une théorie orientable en un système de réécriture utilisant des listes formées des éléments  $A$  et  $B$  et définissant une fonction de tri  $Sort$  supposée ranger tous les  $B$  avant les  $A$ . Pour cela nous supposons qu'il existe deux fonctions  $Inf$  et  $Max$  définissant l'ordre  $A < B$ .

$$\begin{array}{ll}
 Max(A, B) & \rightarrow A \\
 Max(B, A) & \rightarrow A \\
 \\ 
 Inf(A, B) & \rightarrow B \\
 Inf(B, A) & \rightarrow B \\
 \\ 
 Sort(nil) & \rightarrow nil \\
 Sort(Cons(y, nil)) & \rightarrow Cons(y, nil) \\
 Sort(Cons(x, Cons(y, l))) & \rightarrow Cons(Inf(x, y), Cons(Max(x, y), Sort(l)))
 \end{array}$$

<sup>9</sup>puisque  $\alpha_0$  est bijective et n'utilise que des nouveaux états

Si cette fonction de tri était bien définie, alors pour toute liste formée de  $A$  et de  $B$ , aucun  $A$  ne serait avant un  $B$ . Pour vérifier cela, nous proposons de définir un automate initial reconnaissant l'ensemble des termes  $Sort(l)$  où  $l$  est une liste quelconque contenant des  $A$  et des  $B$ .

<b>Automaton A</b>	
<b>States</b>	
$qA \ qB \ qlist \ qnil \ qf$	
<b>Final States</b>	
$qf$	
<b>Transitions</b>	
$nil$	$\rightarrow qnil$
$A$	$\rightarrow qA$
$B$	$\rightarrow qB$
$Cons(qA, qnil)$	$\rightarrow qlist$
$Cons(qB, qnil)$	$\rightarrow qlist$
$Cons(qA, qlist)$	$\rightarrow qlist$
$Cons(qB, qlist)$	$\rightarrow qlist$
$Sort(qlist)$	$\rightarrow qf$

Tout terme  $Sort(l)$  se réécrit donc en un terme formé sur l'alphabet  $\{A, B, nil, Cons\}$ . Donc, si nous choisissons une approximation  $\alpha$  telle que le processus de complétion termine, l'automate obtenu  $\mathcal{A}_{\mathcal{R}, \alpha}^*$  est un automate qui reconnaît un sur-ensemble des listes  $l'$  tel qu'il existe une liste  $l$  vérifiant  $l' = Sort(l)$ . Notre but est de vérifier que cet automate ne reconnaît aucune liste où les  $A$  apparaissent avant les  $B$ , pour détecter toutes les erreurs, il faudrait définir un automate  $A'$  reconnaissant l'ensemble des listes de la forme  $[[AB]^*, A, [AB]^*, B, [AB]^*]$ . Mais il est suffisant d'écrire un automate  $\mathcal{A}'$  reconnaissant les listes de la forme  $[A, A^*, B, B^*]$  de vérifier que  $\mathcal{A}_{\mathcal{R}, \alpha}^* \cap \mathcal{A}'$  est un automate reconnaissant un langage vide.

<b>Automaton A'</b>	
<b>States</b>	
$qA \ qB \ qlistA \ qlistB \ qnil \ qf$	
<b>Final States</b>	
$qf$	
<b>Transitions</b>	
$A$	$\rightarrow qA$
$B$	$\rightarrow qB$
$nil$	$\rightarrow qnil$
$Cons(qB, qnil)$	$\rightarrow qlistB$
$Cons(qB, qlistB)$	$\rightarrow qlistB$
$Cons(qA, qlistB)$	$\rightarrow qf$

Nous proposons d'étudier la complétion de  $\mathcal{A}$  par  $\mathcal{R}$  suivant la fonction d'abstraction  $\alpha_0$  qui normalise chaque sous terme d'une transition à ajouter par un nouvel état. Dans cet exemple, le

processus ne termine pas mais à la seconde étape de complétion, l'intersection entre  $\mathcal{A}_{\mathcal{R},\alpha}^1$  et  $\mathcal{A}'$  est l'automate suivant :

<b>Automaton</b> inter	
<b>States</b>	
$q0$	$q1$ $q2$ $q3$
<b>Final States</b>	
$q4$	
<b>Transitions</b>	
$nil$	$\rightarrow q0$
$B$	$\rightarrow q1$
$A$	$\rightarrow q2$
$Cons(q1, q0)$	$\rightarrow q3$
$Cons(q1, q3)$	$\rightarrow q3$
$Cons(q2, q3)$	$\rightarrow q4$

Cet automate reconnaît l'ensemble des listes de la forme  $[A, B, B, \dots, B]$ , comme la fonction d'approximation n'ajoute pas de nouveaux termes, il existe des listes  $l$  telle que  $Sort(l)$  est de la forme  $[A, B, B, \dots, B]$ . La fonction  $Sort$  telle qu'elle est définie ne trie donc pas les listes correctement<sup>10</sup>.

La complétion d'automate nous permet donc de vérifier des propriétés de la forme  $t\sigma \neq g$  pour tout terme  $t$ , toute substitution  $\sigma$  et tout terme clos  $g$ . L'avantage de l'approche par automate est que la seule restriction sur la théorie considérée est que pour toute équation  $e = e'$  de  $\mathbf{T}$  soit  $Var(e) \subset Var(e')$ , soit  $Var(e') \subset Var(e)$ , il est possible d'utiliser des systèmes non-terminants ou non convergents comme nous le ferons durant l'étude des protocoles cryptographiques.

<sup>10</sup>Une fonction de tri correcte aurait été le point fixe de  $Sort^n$



Deuxième partie

Validation de protocoles  
cryptographiques par réécriture



# 4

## Introduction à la cryptographie

### **Cryptologie**

*Ensemble de techniques qui permettent de protéger des informations grâce à un chiffrement*

### **Cryptographie**

*Etude des algorithmes et des protocoles utilisés pour préserver la confidentialité de l'information et garantir son intégrité*

### **Cryptanalyse**

*Recherche des attaques permettant de casser les protections mises en place*

---

**Sommaire**


---

<b>4.1 Un peu de cryptographie</b> . . . . .	<b>46</b>
4.1.1 la cryptographie à clé secrète . . . . .	46
4.1.2 la cryptographie à clé publique . . . . .	48
<b>4.2 Utilisation de la cryptographie</b> . . . . .	<b>51</b>
4.2.1 Buts actuels . . . . .	51
4.2.2 Quelles sont les menaces ? . . . . .	51
<b>4.3 Vérification de protocoles cryptographiques</b> . . . . .	<b>51</b>
4.3.1 Modélisation classique . . . . .	51
<b>4.4 Modélisation proposée : généralités</b> . . . . .	<b>53</b>

---

## 4.1 Un peu de cryptographie

La cryptographie est l'ensemble des techniques qui permettent de dissimuler des informations à l'aide d'algorithmes de chiffrement. Un protocole cryptographique est un ensemble de règles concernant des échanges de messages entre plusieurs participants qui doit mener à un accord ou au partage d'une donnée secrète. Les problèmes viennent du fait que ces messages circulent sur des réseaux de communication a priori non fiables. Après avoir donné un aperçu de deux techniques de cryptographie, à savoir la cryptographie dite *symétrique* ou à clé secrète et *asymétrique* ou à clé publique, nous nous proposons ici d'étudier les utilisations et les problèmes des protocoles cryptographiques. Pour terminer ce chapitre nous proposerons un bref état de l'art des diverses méthodes de vérification avant de présenter brièvement une méthode de vérification que nous développerons plus amplement dans le chapitre suivant pour modéliser et vérifier le protocole *copy protection* du système **SmartRight**.

### 4.1.1 la cryptographie à clé secrète

*Le principe de la cryptographie à clé secrète consiste à utiliser une même clé pour chiffrer et déchiffrer les informations.*

Lorsqu'un groupe d'utilisateurs utilise la cryptographie pour sécuriser ses échanges, il peut choisir de partager une clé secrète. Si un même utilisateur souhaite communiquer avec des groupes différents, il doit posséder les clés correspondantes, ce qui conduit à utiliser un grand nombre de clés aussitôt que ce schéma s'étend à de nombreux groupes d'utilisateurs. La gestion, la garantie de confidentialité et d'intégrité des clés secrètes engendre alors des difficultés en termes de ressources et d'organisation.

La cryptographie à clé secrète repose sur deux algorithmes : l'un utilise une clé pour chiffrer un message, et le second effectue l'opération inverse à l'aide de la même clé. L'algorithme en lui-même est public, et la clé concentre l'intégralité du secret. Pour échanger des messages il faut tout d'abord envoyer la clé à son correspondant ce qui pose le problème de la sécurité des échanges de clés. Nous donnons ici un algorithme de cryptographie à clé secrète : l'algorithme D.E.S..

### Un algorithme à clé secrète : D.E.S.

D.E.S. (Data Encryption Standard ) a été crée à l'origine par IBM en 1976 et est réactualisé tous les cinq ans ; la dernière réactualisation a été faite en 1994. Passé cette date D.E.S. ne sera

plus réactualisé.

D.E.S. est un système de chiffrement par blocs de 64 bits. D.E.S. utilise une clé secrète de 56 bits, qu'il transforme en 16 "sous-clés" de 48 bits chacune. Le cryptage se déroule sur 19 étapes. Il s'agit de faire des combinaisons, des substitutions et des permutations entre le texte à chiffrer et la clé, toutes les opérations devant être bijectives pour pouvoir réutiliser cette même clé lors du déchiffrement.

Étapes :

1. fractionnement du texte clair en blocs de 64 bits

1	2	3	4	5	6	7	8
9	10	11	12	13	14	15	16
17	18	19	20	21	22	23	24
25	26	27	28	29	30	31	32
33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48
49	50	51	52	53	54	55	56
57	58	59	60	61	62	63	64

2. Permutation initiale au sein des blocs du texte clair, notée P

33	29	55	61	20	4	62	40
54	10	5	38	23	6	19	7
9	2	11	42	40	53	34	16
21	58	46	60	3	8	1	45
41	12	31	44	37	59	32	48
17	26	39	37	18	22	36	24
25	57	35	43	15	30	28	52
49	50	51	64	14	63	27	56

3. Découpage des blocs en partie gauche et partie droite : on obtient deux suites L(i) et R(i) de blocs à 32 bits.

Les étapes 3 et 4 sont répétées 16 fois

4. On applique l'algorithme suivant :

$$L(i) = R(i - 1)$$

$$R(i) = L(i - 1) + f(R(i - 1), K(i))$$

La fonction f applique d'abord à R(i-1)

- une permutation expansive à 48 bits : les 32 bits de R(i-1) entrent dans une table de sélection de bits E, où ils sont mélangés et répétés, afin d'obtenir 48 bits. Les 48 bits de B' sont transformés par OU-Exclusif avec  $K_i : B' \longrightarrow B' \oplus K$ .
- puis, le résultat de l'étape précédente est divisé en 8 blocs  $B_i$  pour  $i \in \{1, \dots, 8\}$  de 6 bits chacun : (b1,b2,b3,b4,b5,b6).
- La décomposition permettra de déterminer une position dans une table de sélection S à blocs de 16 colonnes et 4 lignes : Le nombre binaire (b1 b6) représente le numéro de ligne x, le nombre binaire (b2 b3 b4 b5) le numéro de colonne y. Une fois la position (x,y) trouvée, on substitue au bloc  $B_i$  le bloc de 4 bits déterminé par son adresse (x,y) dans la table. Ce qui nous donne un résultat de 32 bits.
- enfin une permutation .

Le  $\oplus$  est un ou-exclusif .

5. On recolle les parties gauches et droites des blocs puis on effectue enfin l'inverse de la permutation initiale P.

D.E.S. est un système de chiffrement à 16 étapes (ou *rondes*). D.E.S. était initialement à quatre rondes, mais aujourd'hui grâce aux méthodes de cryptanalyse de Biham et Shamir, il est possible de déchiffrer D.E.S. jusqu'à 15 rondes , d'où le nombre 16 .

Pour décrypter le message il suffit de refaire le même algorithme que ci-dessus en appliquant d'abord la seizième clé puis la quinzième etc... jusqu'à la première.

D.E.S. n'est actuellement plus fiable car la puissance des machines permet une recherche exhaustive, c'est pourquoi on utilise notamment DES-3, qui consiste à crypter trois fois de suite un message, en utilisant deux clés différentes. L'International Data Encryption Algorithm (IDEA) quant à lui a été développé par X. Lai et J. Massey en 1991 pour remplacer le D.E.S. Son principe est le même que le DES sauf qu'il utilise une clé de 128bits, il a été utilisé par P.Zimmerman dans PGP (Pretty Good Privacy) qui est une combinaison d'algorithmes de cryptographie symétriques et asymétriques. Le principal avantage de ces algorithmes est leur rapidité, ce qui explique en partie le grand nombre d'implémentations de ces derniers dans des puces électroniques pour les besoins des entreprises. Le principal inconvénient est l'échange des clés, il faut réussir à transmettre la clé à son correspondant sans qu'elle soit interceptée, ce qui n'est pas le cas de la cryptographie à clé publique.

#### 4.1.2 la cryptographie à clé publique

*La cryptographie à clé publique utilise deux clés distinctes pour le chiffrement et le déchiffrement. Toutefois, le couple de clés utilisé est cryptographiquement indissociable : il comprend une partie privée qui est secrète et une partie publique, dont la confidentialité n'est pas nécessaire. Ce couple de clés est appelé bi-clé.*

La cryptographie à clé publique permet de résoudre les problèmes d'échanges de clés de la cryptographie à clé secrète, dans la mesure où chaque utilisateur dispose d'une clé privée et d'une seule. Ainsi, lorsqu'un utilisateur souhaite établir une convention secrète, il utilise la clé publique de son interlocuteur et met en oeuvre un mécanisme cryptographique asymétrique.

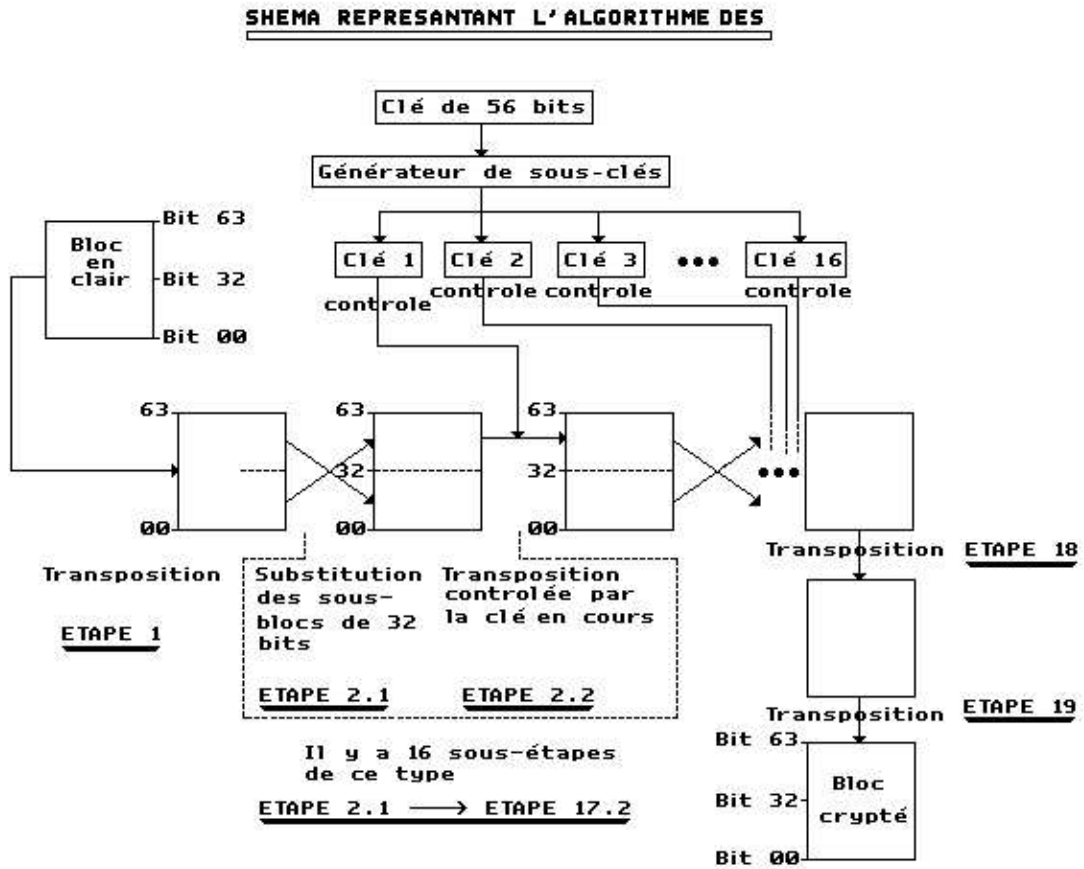
Dans la cryptographie à clé publique, les clés de chiffrement et de déchiffrement sont différentes, et il est extrêmement difficile de deviner l'une d'elles en fonction de l'autre : pour fabriquer un cryptosystème à clé publique, on utilise un problème tel qu'on ne connaisse pas d'algorithme permettant de le résoudre en un temps raisonnable. Il est alors possible de publier une des deux clés (la clé publique) afin d'autoriser le chiffrement à tous les intervenants en conservant la clé de déchiffrement (la clé privée) secrète. Ainsi, il est possible d'envoyer un message chiffré à un correspondant sans avoir échangé une clé secrète avec lui. Nous proposons comme exemple d'algorithme cryptographique à clé publique, l'algorithme RSA, inventé par Rivest, Shamir, et Adleman en 1978, l'algorithme de 1978 RSA128 utilisait une clé de 128 bits et n'a été cassé qu'en 1996. Aujourd'hui RSA s'utilise avec des clés de 1024, 2048 voire 4096 bits.

#### Un algorithme à clé publique : l'algorithme RSA

Prenons  $p$  et  $q$  deux nombres premiers, nous savons d'après le petit théorème de Fermat que

$$\forall m \text{ tel que } \text{pgcd}(m, p) = 1 \quad m^{p-1} \equiv 1 \quad [p]$$

$$\forall m \text{ tel que } \text{pgcd}(m, q) = 1 \quad m^{q-1} \equiv 1 \quad [q]$$



source <http://histoirecrypto.ifrance.com/histoirecrypto/>

FIG. 4.1 – schéma de l'algorithme DES

d'après la généralisation d'Euler de ce théorème,

$$m^{(p-1)(q-1)} \equiv 1 \pmod{pq}$$

choisissons  $e$  tel que

- $2 < e < (p-1)(q-1)$
- $e$ , et  $(p-1)(q-1)$  soient premiers entre eux

et choisissons  $d$  tel que  $ed \equiv 1 \pmod{(p-1)(q-1)}$ .

Posons  $n = pq$  et  $\phi(n) = (p-1)(q-1)$ , le couple  $(e, n)$  est la clé publique : considérons que  $m$  soit le message non crypté, le message crypté  $c$  est défini par la relation :  $c = m^e$ .

Le couple  $(d, n)$  est la clé privée : pour retrouver  $m$  à partir de  $c$  il suffit de calculer  $c^d \equiv m \pmod{n}$ , en effet

$$\begin{aligned} c^d &= (m^e)^d \\ &= m^{ed} \\ &\equiv m^{1+k \times \phi(n)} \pmod{n} \text{ pour un certain } k \text{ dans } \mathbb{N} \\ &\equiv m \times m^{k \times \phi(n)} \pmod{n} \\ &\equiv m \times (m^{\phi(n)})^k \pmod{n} \\ &\equiv m \pmod{n} \end{aligned}$$

Une fois les entiers  $p$  et  $q$  détruits, il est extrêmement difficile de retrouver  $d$  à partir de  $(e, n)$ . Il n'existe pas pour le moment d'algorithme de complexité inférieure à  $e^{n \times \log n}$  permettant de trouver la factorisation d'un nombre de  $n$  chiffres. Ainsi au début de 1999 un nombre de 140 chiffres, qui était le produit de deux nombres premiers de 70 chiffres chacun a été factorisé en environ un mois de calculs à l'aide d'un imposant parc de machines.

RSA a un taux de chiffrement de 600 Ko par seconde. Comparé aux 1 Go par seconde de DES, RSA est 1500 fois plus lent que DES. La transmission de gros fichiers est donc particulièrement lente, ce qui n'est guère pratique. Ceci est un des principaux défauts de RSA. Malgré tout, RSA offre deux propriétés essentielles :

- Si l'on chiffre un message avec la clé privée, non seulement seule la clé publique correspondante permet de retrouver le message initial, mais en outre, le déchiffrement avec une autre clé donne un message complètement différent. Cette propriété garantit que la clé privée utilisée à l'origine du message traité est celle de son possesseur unique et de personne d'autre : l'algorithme offre donc une garantie d'identité.
- La seconde propriété est que toute modification, même très faible, du message chiffré à l'aide de la clé privée le rend totalement différent et donc totalement incompréhensible lorsqu'on le déchiffre avec la clé publique. Cette propriété permet donc de garantir l'intégrité du message.

Ainsi, l'algorithme RSA permet de garantir à la fois l'identité de l'émetteur du message, et son intégrité.

La partie qui suit va mettre en évidence les divers besoins actuels en cryptographie ainsi que les problèmes rencontrés, cette partie s'inspire du site du Ministère de l'Economie, des Finances et de l'Industrie [http://www.telecom.gouv.fr/secur/crypt\\_util.htm](http://www.telecom.gouv.fr/secur/crypt_util.htm).

## 4.2 Utilisation de la cryptographie

### 4.2.1 Buts actuels

Il est essentiel pour le développement des échanges électroniques et de la société de l'information, que les membres de la communauté aient la possibilité :

- D'avoir l'assurance que leurs interlocuteurs sont bien ceux qu'ils croient (**Authentification**)
- D'assurer la confidentialité de leurs échanges (**Confidentialité**)
- De disposer des informations au moment où ils en ont besoin (**Disponibilité**)
- Que les messages envoyés et les messages reçus sont bien ceux qu'ils ont envoyés ou que leurs interlocuteurs ont envoyés (**Intégrité**)
- D'avoir la preuve de l'émission d'une information ou de sa réception. L'émetteur ou le récepteur ne peut ainsi en nier l'envoi ou la réception (**Non Répudiation**)

### 4.2.2 Quelles sont les menaces ?

On fait en général la distinction entre des attaques passives et des attaques actives. Les attaques passives consistent en l'interception et l'exploitation de l'information. Elles ne modifient pas les informations échangées, ce sont des attaques à la confidentialité. Les attaques actives constituent un champ beaucoup plus large et complexe d'activités criminelles : il peut s'agir de couper, ralentir ou dégrader la communication, de saturer des systèmes avec des informations parasites, de modifier les informations à l'insu du destinataire, de créer de fausses informations, de les aiguiller vers un destinataire autre que le destinataire légitime, ou de les faire carrément disparaître. La protection contre les attaques actives passe par l'assurance de la disponibilité de l'information là où elle doit être utilisée ou transportée, et par l'assurance de son intégrité. Malheureusement la détection des failles est en général indécidable, malgré tout, il apparaît aujourd'hui nécessaire d'offrir des garanties de sécurité sur les moyens cryptographiques employés, c'est le but des méthodes formelles.

## 4.3 Vérification de protocoles cryptographiques

Notre lecteur trouvera une documentation sur la vérification de protocoles cryptographiques bien plus riche dans [Cor03].

### 4.3.1 Modélisation classique

Un protocole cryptographique est en général représenté comme un ensemble de transitions de la forme

$$A \leftrightarrow B : \{M\}_k$$

qui signifie *A envoie à B le message M chiffré avec la clé k*. Un modèle classique qui sert de base à la plupart des méthodes de vérification est celui dû à Dolev et Yao [DY83]. Dans cette modélisation, l'intrus est omnipotent, on considère qu'il peut écouter, intercepter, et/ou corrompre tous les messages qui circulent sur n'importe quelle ligne de communication. De ce fait il n'est pas nécessaire de modéliser les canaux de communication.

Par ailleurs dans la modélisation de Dolev Yao les messages envoyés sont représentés sous la forme d'une algèbre de termes : Un message M peut être

- une donnée de base (entier, réel, booléen)
- une clé  $k$
- un couple de messages :  $\{M_1, M_2\}$
- le premier élément d'un couple  $p_1(M)$
- le second élément d'un couple  $p_2(M)$
- un message  $M$  chiffré par la clé  $k$  :  $\{M\}_k$
- un message en clair déchiffré avec la clé  $k$  :  $\{\{M\}_k\}_k$

La dernière partie de la modélisation de Dolev Yao consiste en la représentation de la connaissance de l'intrus à partir de règles de déduction opérant sur l'ensemble de messages  $E$  et définissant les messages que l'intrus peut construire :

- Un intrus connaît tous les messages de  $E$  :  $E, M \rightsquigarrow M$
- Un intrus peut former des couples :  $E, M_1, M_2 \rightsquigarrow \{M_1, M_2\}$
- Un intrus peut récupérer le premier (resp. le second) élément d'un couple :  $E, M \rightsquigarrow p_1(M)$   
( $E, M \rightsquigarrow p_2(M)$ )

Les deux méthodes de vérification suivantes s'appuient sur un modèle des protocoles cryptographiques dérivé de celui ci.

### Vérification à l'aide d'assistants de preuve

Les méthodes de vérification à l'aide d'assistants de preuve comme COQ [Bol96] ou Isabelle [Pau97] s'appuient sur une modélisation des protocoles en vérification en logique du premier ordre. Lawrence C. Paulson propose dans [Pau97] une méthode inductive pour la vérification des protocoles. Chaque configuration du protocole est un "événement", une séquence d'événements constitue une trace. Chaque étape du protocole est définie par induction et définit une extension de trace, c'est à dire que les actions permises par le protocole ainsi que les actions de l'intrus sont codées par l'ajout d'un événement à une trace. La vérification du protocole consiste à vérifier des propriétés par induction sur l'ensemble des traces à l'aide l'assistant de preuve Isabelle.

D. Bolignano développe une méthode qui consiste tout d'abord en une représentation de chaque *agent* ou *principal* par un automate dont les états sont des  $n$ -uplets qui codent la configuration dans laquelle se trouve l'agent à chaque étape du protocole. Le rôle d'un agent est l'ensemble de actions décrites par le protocole; chacune de ces actions sera représentée par une relation ou prédicat sur des couples d'états des automates de chaque agent. L'intrus est représenté par un ensemble  $E$  de messages qu'il est capable de récupérer,

### Modélisation par algèbre de processus

M. Abadi et A.D. Gordon ont utilisé le Spi-calcul, un modèle de programmes dérivé du Pi-calcul dans le but de vérifier des propriétés d'authentification sur les protocoles. Le Pi-calcul permet de raisonner sur les systèmes distribués communicants, il est proche du langage CSP (Communicating Sequential Processes). Il permet de décrire des systèmes dont le nombre de processus ainsi que les liens de communication entre processus peuvent varier au cours du temps (processus mobiles). Le Spi-calcul possède en plus des primitives cryptographiques, il permet d'exprimer l'envoi et la réception de messages, la création de nonces<sup>11</sup>, la duplication de messages. Les protocoles seront représentés par des systèmes où les processus seront les agents du

---

<sup>11</sup>Un nonce est un nombre aléatoire qui est utiliser pour marquer la *fraisheur* des message, par exemple on peut choisir de creer un nonce par session, ...

protocole. Les propriétés de sécurité d'un protocole seront représentées par des équivalences entre systèmes. Considérons par exemple une valeur  $X$ , où  $X$  est un secret si un observateur ne peut pas distinguer de différence entre le comportement du système connaissant  $X$  et un système dans lequel  $X$  est remplacé par n'importe quelle valeur  $X'$ . La notion de secret est ici plus forte que celle du modèle précédent, ici un message est secret si et seulement si l'intrus ne peut rien en dire.

### Approche explorative

Pour conclure ce bref panorama des diverses méthodes de vérification des protocoles cryptographiques, il nous reste à présenter l'approche de C.Meadows implémentée dans l'outil NRL [Mea96]. NRL est basé sur une modélisation sous forme de règles de réécriture du modèle Dolev Yao. Le principe est de spécifier l'ensemble des termes que l'intrus ne doit pas obtenir comme les clés privées des agents honnêtes, les messages chiffrés, etc... Ensuite on essaie d'appliquer les règles de réécritures disponibles à l'envers (recherche arrière ou backward search) afin de remonter à un état initial. Cette opération est bien entendu généralement non terminante puisque l'ensemble des états est potentiellement infini. Afin de restreindre l'espace de recherche, l'outil offre la possibilité à l'utilisateur de guider la recherche. Par ailleurs, il est possible de montrer que des ensembles infinis d'états sont inatteignables à l'aide de techniques inductives. L'algorithme utilisé ne permet pas de conclure qu'un langage est atteignable si le processus échoue, par contre si le processus réussit, le langage étudié est effectivement inatteignable.

## 4.4 Modélisation proposée : généralités

Nous proposons une méthode de vérification basée sur la preuve de non-atteignabilité de configurations interdites. Nous proposons de modéliser un protocole sous forme d'un ensemble régulier de termes reconnu par un automate d'arbre  $\mathcal{A}$ , puis de spécifier les actions du protocole et le comportement de l'intrus par des règles de réécriture  $\mathcal{R}$ . L'ensemble  $\mathcal{R}^*(\mathcal{A})$  des termes atteignables par  $\mathcal{R}$  à partir de  $\mathcal{A}$  est exactement l'ensemble des configurations permises par le protocole dans un environnement hostile décrit par les règles de l'intrus. Nous ne proposerons pas de modèle de vérification, nous pensons que chaque modélisation dépendra du protocole et des propriétés à vérifier. Notre modèle de l'intrus s'inspirera du modèle de Dolev-Yao. Dans le chapitre suivant nous présentons notre méthode par l'exemple de l'étude d'un protocole développé pour la protection du contenu vidéo de Thomson.



5

# Application à la vérification de protocoles

## Sommaire

<b>5.1</b>	<b>Vérification du protocole <i>View Only</i> de SmartRight</b>	<b>56</b>
5.1.1	Présentation du protocole	56
5.1.2	Spécification du protocole	59
5.1.3	Les configurations initiales	61
<b>5.2</b>	<b>SmartRight et l'intrus</b>	<b>63</b>
5.2.1	Rejeu	63
5.2.2	Un intrus capable de chiffrer, déchiffrer, composer et décomposer	65
5.2.3	Un intrus connaissant $\mathcal{K}$	66
5.2.4	Un intrus générant des <i>Control Word</i>	67
<b>5.3</b>	<b>Conclusion</b>	<b>67</b>

---

## 5.1 Vérification du protocole *View Only* de SmartRight

Dans ce chapitre nous étudierons un protocole développé par *Thomson Multimédia* dans le système **SmartRight** qui vise la protection du contenu numérique. De façon générale, ce système se propose de centraliser tous les droits à l'image et au son de l'environnement numérique d'un même foyer : décodeurs et postes de TV numériques, ordinateurs, appareils internet de salon, etc... Le contenu numérique (image/son) sera crypté sur l'ensemble du réseau domestique, puis décodé par la carte à puce uniquement au moment de la lecture. Il sera possible d'enregistrer une vidéo, la regarder en différé et conserver des copies, ou échanger ce contenu mais uniquement entre les différents appareils du même espace domestique. Les fournisseurs de ces contenus pourront fixer librement les options de protection, comme « lecture seule », « copie privée » ou « copie libre ». Nous étudierons le protocole *View Only* de **SmartRight** qui gère l'option de protection « lecture seule ».

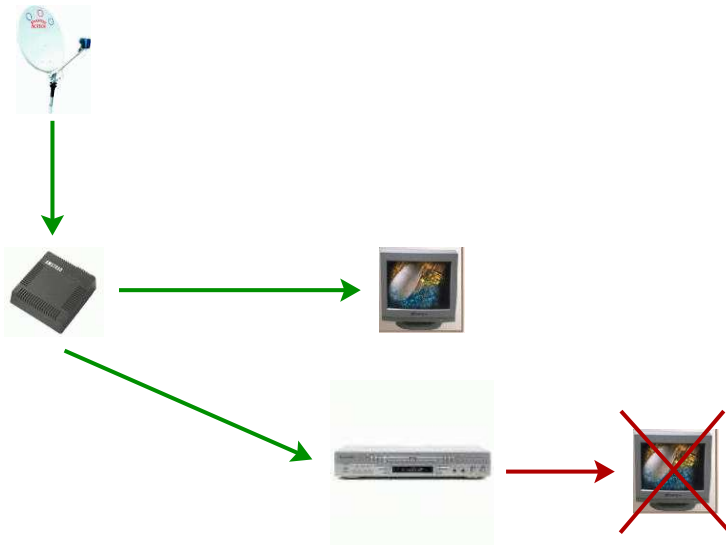
### 5.1.1 Présentation du protocole

#### But du protocole

Le protocole *View Only* de **SmartRight** se joue entre deux cartes à puce respectivement nommées *Converter Card* et *Terminal Card*. La première carte : *Converter Card* équipe l'appareil de réception du flux vidéo à l'intérieur de l'espace domestique, la seconde carte : *Terminal Card* équipe un appareil numérique comme une télévision, un magnétoscope, ... La première carte est responsable de la communication d'une part du flux vidéo chiffré et d'autre part du flux de clés appelées *Control Word* nécessaire à la seconde carte pour le déchiffrement de la vidéo. Pour une vidéo donnée, il n'existe qu'un seul flux de *Control Word* permettant son déchiffrement. Enfin n'importe quel *Terminal Card* est capable de déchiffrer une vidéo s'il possède les *Control Word* correspondants. L'option « lecture seule » du système **SmartRight** gérée par le protocole *View Only* est d'assurer qu'une vidéo ne puisse être lue qu'une seule fois (voir figure 5.1.1).

#### Moyens cryptographiques

Les cartes à puce d'un même espace domestique possèdent toutes la même clé secrète que nous noterons  $\mathcal{K}$ . A l'initialisation du réseau domestique, une seule carte connaît  $\mathcal{K}$ , cette carte doit alors communiquer la clé  $\mathcal{K}$  aux autres cartes du réseau. La distribution de la clé  $\mathcal{K}$  se fait

FIG. 5.1 – Le protocole *View Only* de **SmartRight**

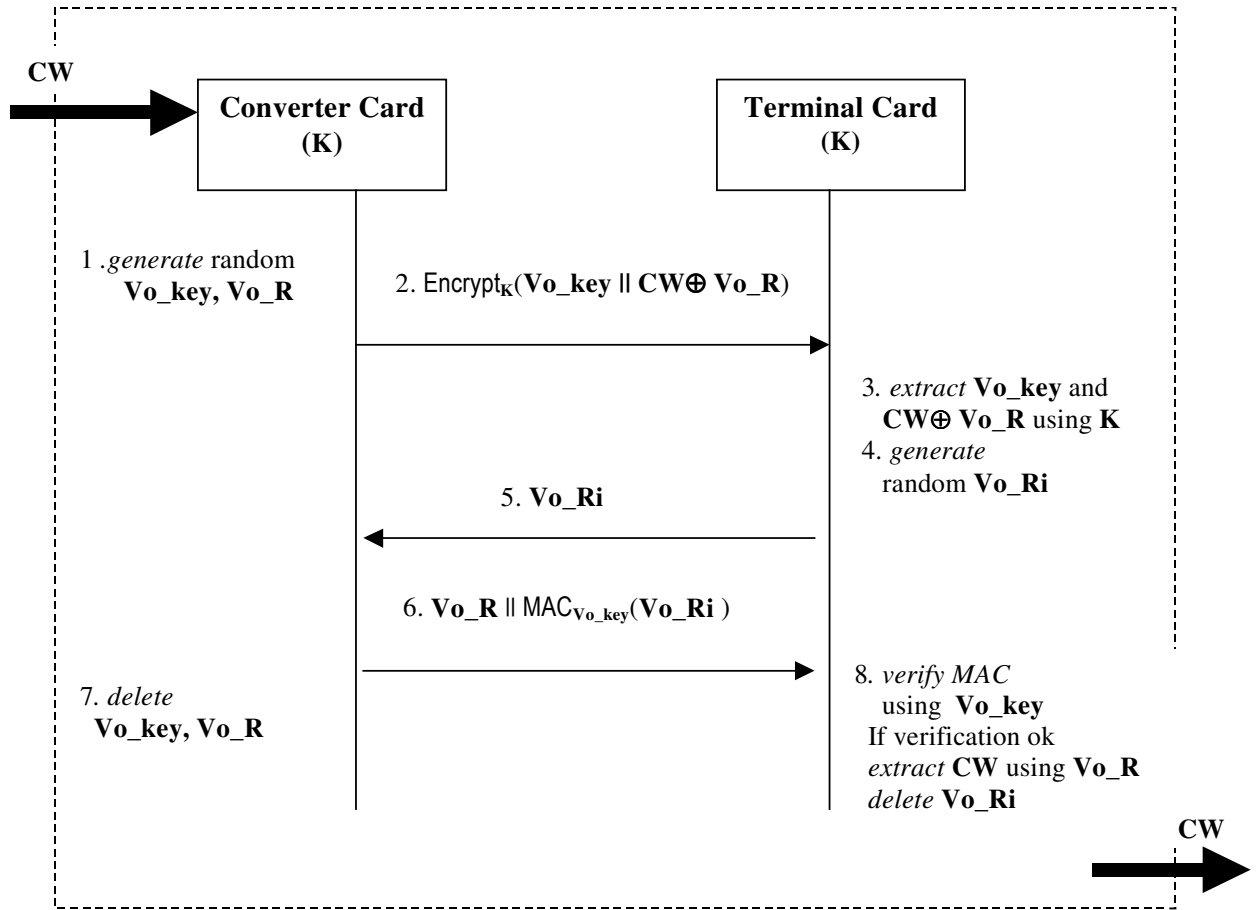
en nombre limité. Le partage de cette clé est un point crucial du système **SmartRight**, la principale hypothèse de notre vérification est que l'intrus n'a pas la connaissance de cette clé. Nous montrerons que si cette hypothèse n'est pas vérifiée alors aucune propriété n'est assurée. En dehors de cette clé, le *Converter Card* est capable de générer de manière aléatoire deux flux de clés respectivement appelées  $Vo\_R$  et  $Vo\_key$ . La carte à puce d'un *Terminal Card* utilise un dernier seul flux de clés appelées  $Vo\_Ri$ . Pour sécuriser leurs échanges, ces cartes vont encore utiliser la fonction *Xor* ainsi qu'une fonction de hachage combinée à un chiffrement :  $Mac(m, k)$  représente le résultat du hachage du message  $m$  chiffré par la clé  $k$ . Ces trois familles de clés ( $Vo\_R$ ,  $Vo\_key$ ,  $Vo\_Ri$ ), la fonction *Xor* et la fonction *Mac* forment l'ensemble des moyens cryptographiques utilisés par le protocole *View Only*.

### Propriétés à vérifier

Nous dirons qu'une carte à puce est *honnête* si elle appartient à l'espace domestique autrement elle sera considérée comme *intruse*. Pour assurer qu'une vidéo ne soit lue qu'une seule fois, nous allons vérifier que :

1. les *Control Word* émis sont reçus et dans le bon ordre
2. un même *Control Word* est reçu au plus une fois
3. une carte à puce *intruse* ne peut connaître aucun *Control Word honnête*
4. une carte à puce *honnête* n'accepte aucun *Control Word* émis par une carte *intruse*

les deux premières propriétés assurent le bon fonctionnement du protocole, la seconde propriété code la fonctionnalité principale du protocole, elle sera nommée *Anti-Replay*. La troisième propriété est une propriété de secret classique : les informations protégées restent inconnues de l'intrus. La dernière est une propriété d'authentification : elle assure qu'une carte à puce sait distinguer les informations envoyées par une carte appartenant au même espace qu'elle.



### 5.1.2 Spécification du protocole

La spécification du protocole *View Only* de **SmartRight** telle qu'elle nous a été fournie par Thomson est donnée en figure 5.1.1. Nous commentons ici cette spécification avant de proposer une traduction sous forme de termes et système de réécriture que nous utiliserons pour la vérification du protocole. Comme annoncé en 4.4, nous n'avons pas de codage prédéfini pour le protocole, nous proposons simplement de traduire la spécification le plus fidèlement possible sous forme de termes et de règles de réécriture. Plus précisément nous proposons d'étudier le fonctionnement du protocole de façon à savoir décrire une configuration générale du protocole afin de la traduire sous forme d'un ensemble de termes ; parallèlement nous traduirons les diverses étapes du protocole ainsi que le comportement de l'intrus par des règles de réécriture sur ces termes. De façon générale, une configuration quelconque du protocole sera représentée par un terme de la forme  $\text{State}(X, CC, TC, I)$  (Notons qu'à partir de maintenant, les termes en majuscules sont des variables et réciproquement) où  $X$  est le message courant circulant sur le réseau,  $CC$ ,  $TC$  et  $I$  représentent respectivement le *Converter Card*, le *Terminal Card* et l'intrus.

#### Initialisation et étape 1

A l'initialisation du protocole, le *Converter Card* attend de recevoir un *Control Word*, lorsqu'il le reçoit, il génère deux nouvelles clés :  $Vo\_R$  et  $Vo\_key$ . Pour modéliser sous forme de termes que les clés sont uniques, et sont changées à chaque fois que le *Converter Card* reçoit un *Control Word*, nous les modéliserons comme des fonctions du *Control Word* courant :  $\text{key}(\text{vor}(CW))$  et  $\text{key}(\text{vokey}(CW))$ . Le *Converter Card* envoie alors un premier message :  $\{Vo\_key, \text{xor}(CW, Vo\_R)\}_{\mathcal{X}}$ . Notre modélisation de l'intrus n'aura pas accès à la valeur du paramètre des clés ce qui fait que cette modélisation est une approximation correcte de la réalité au sens où les clés seront nouvelles pour chaque réception d'un *Control Word* et leurs valeurs apparaîtront aléatoires à l'intrus. Nous symboliserons le flux de *Control Word* par une liste infinie de *Control Word*. Dans cette représentation, nous modélisons une liste de *Control Word* dans laquelle nous mettons en valeur un élément particulier :  $\text{current\_cw}$ . Les *Control Word* précédents (resp. suivants) sont indistingables du point de vue de l'automate. Autrement dit, cette modélisation décrit l'ensemble le langage de *Control Word* suivant :  $[\text{old\_cw}^*, \text{current\_cw}, \text{next\_cw}^*]$ .

- $\text{nil}$  si la liste est vide
- $\text{cons}(\text{cw}, 1)$  si la liste contient au moins un élément  $\text{cw}$  placé en tête.

Dans l'automate, cette liste sera représentée par les transitions suivantes :

**Transitions**

```

        nil → qlistnext
        old_cw → qold
        current_cw → qcurrentcw
        next_cw → qnextcw
        Key(qoldcw) → qold
        Key(qcurrentcw) → qcurrent
        Key(qnextcw) → qnext
        cons(qnext, qlistnext) → qlistnext
        cons(qcurrent, qlistnext) → qlistcurrent
        cons(qold, qlistcurrent) → qlistold
        cons(qold, qlistold) → qlistold
    
```

Nous modéliserons le *Converter Card* comme une fonction de trois paramètres :

`convertercard(CWL, VK, VR)`

où `CWL` représente le flux de *Control Word* entrant, tandis que `VK` et `VR` seront les valeurs des clés *Vo\_key* et *Vo\_R*.

Finalement, cette première étape se code par la règle de réécriture suivante :

```

State(X, convertercard(cons(CW, Y), Z, U), TC, I)
->
State(encrypt(Key(k), cons(key(vokey(CW)), cons(Xor(CW, key(vor(CW))), nil))),
      convertercard(cons(CW, Y), Key(vokey(CW)), Key(vor(CW))), TC, I)
    
```

**Etapes 1 - 2**

Un message chiffré par  $\mathcal{K}$  circule maintenant sur le réseau, ce qui déclenche l'initialisation du protocole pour le *Terminal Card*; celui-ci récupère le message, le déchiffre, stocke l'information en clair et génère une nouvelle clé *Vo\_Ri*. Cette clé est modélisée sous forme de fonction du *Control Word* courant ainsi que de la valeur courante de la clé *Vo\_key* reçue. Nous choisissons de représenter le *Terminal Card* sous la forme d'une fonction de 3 paramètres : son information courante, la valeur de la clé *Vo\_Ri* et la liste des *Control Word* sortants déjà acceptés. Les deux premiers paramètres sont les informations nécessaires au *Terminal Card* pour jouer le protocole, le dernier paramètre est ajouté pour la vérification, c'est encore une fois une approximation correcte de la réalité dans le sens où aucune règle ne va tenir compte de la valeur de la liste des *Control Word* acceptés, en revanche nous pourrions vérifier que cette liste contient bien les valeurs attendues.

Nous aboutissons à la règle suivante :

```

State(encrypt(Key(k), cons(E1, cons(E2, nil))),
      CC, terminalcard(X, Y, CWLIST), I)
->
State(Key(vori(E1, CWLIST)), CC,
      terminalcard(cons(E1, cons(E2, nil)), Key(vori(E1, CWLIST)), CWLIST), I)
    
```

**Etapas 2 - 3**

Un message non chiffré contenant une clé circule sur le réseau, c'est la réponse qu'attend le *Converter Card* pour envoyer son second message : un message en deux parties : d'une part sa clé  $Vo\_R$  en clair et d'autre part le résultat du hachage de la clé qu'il vient de lire sur le réseau chiffré par sa clé  $Vo\_key$ , ce faisant le *Converter Card* efface ses clés  $Vo\_key$  et  $Vo\_R$ . Ce qui se traduit par la règle suivante :

```
State(key(VORI), convertercard(cons(CW, Y), Key(VK), Key(VR)), TC, I)
->
State(cons(key(VR), cons(encrypt(key(VK), hach(Key(VORI))), nil)),
      convertercard(Y, nil, nil), TC, I)
```

**Etapas 3 - 4**

Un dernier message apparaît maintenant sur le réseau, le *Terminal Card* le récupère, déchiffre la seconde partie à l'aide de  $Vo\_key$  reçue à l'étape précédente et compare le message obtenu avec le résultat du hachage de la clé  $Vo\_Ri$  générée précédemment ; si la comparaison est satisfaisante alors il utilise la première partie du message pour récupérer le *Control Word* reçu auparavant. Ces conditions de comparaison se traduisent par des variables non linéaires. Les sous termes de la forme  $read(x, read(y, z))$  représente une liste de *Control Word* dont les deux premiers éléments sont  $x$  et  $y$  qui ont été acceptés par les *Terminal Card*.

```
State(cons(VR, cons(encrypt(VK, hach(VORI)), nil)), CC,
      terminalcard(cons(VK, cons(xor(CW, VR), nil)), VORI, CWLIST), I)
->
State(nil, CC,
      terminalcard(cons(VK, cons(xor(CW, VR), nil)), VORI, read(CW, CWlist)), I)
```

L'ensemble des règles que nous venons de définir, forme une description de toutes les actions permises par le protocole, nous noterons ce système de réécriture  $\mathcal{R}_{proto}$ .

**5.1.3 Les configurations initiales**

Il s'agit maintenant de modéliser l'ensemble des configurations initiales sous forme d'un ensemble régulier de termes qui se codera par un automate d'arbre  $\mathcal{A}$  sur lequel s'appliquera le système de réécriture que nous venons de définir. Dans un premier temps nous étudierons le fonctionnement du protocole en utilisant un ensemble de termes décrivant des situations initiales permises ; tant que le système de réécriture ne modélise pas d'intrus,  $\mathcal{R}_{proto}^*(\mathcal{A})$  est l'ensemble des configurations accessibles grâce au protocole à partir des situations décrites par  $\mathcal{A}$ . Le fait de commencer la vérification sans modélisation de l'intrus va permettre de s'assurer que le protocole a bien le comportement attendu et que la modélisation est fonctionnelle.

Nous proposons une situation initiale où

1. il n'y a pas encore de message sur le réseau
2. le *Converter Card* possède une liste de *Control Word* qu'il doit envoyer mais n'a aucune clé en mémoire

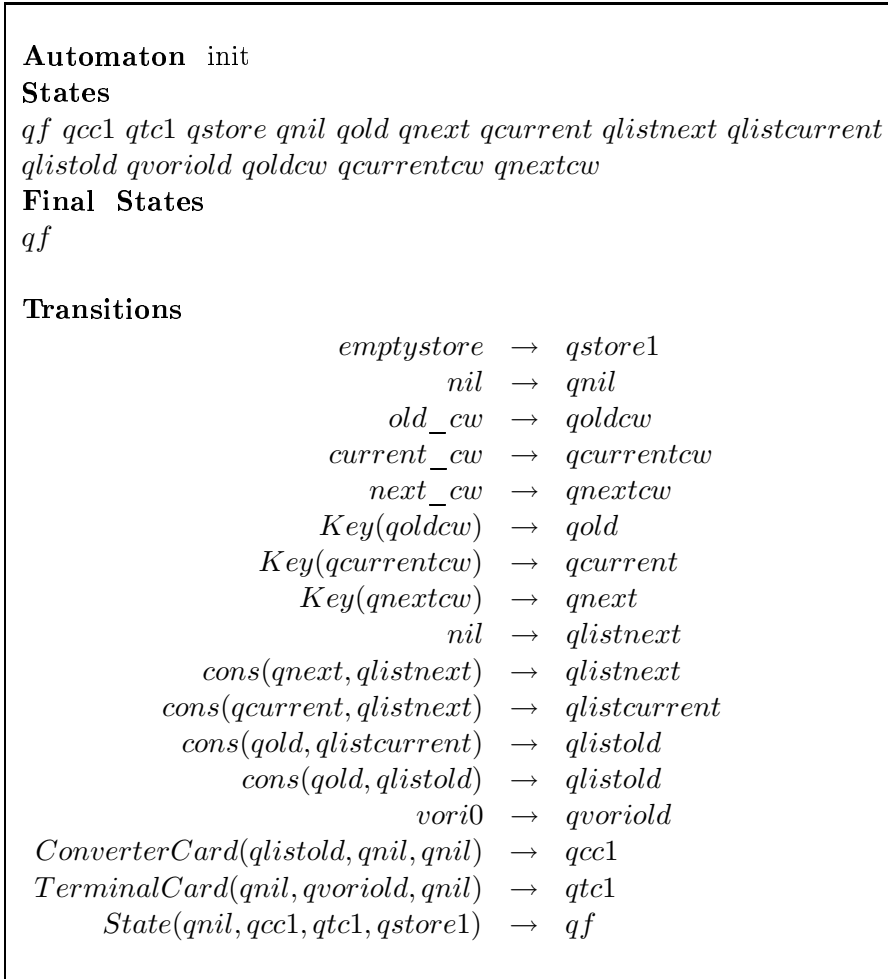


FIG. 5.3 – Automate des configurations initiales

3. le *Terminal Card* n'a pas d'information courante et a une ancienne clé  $\mathcal{Vo\_Ri}$  en mémoire
4. l'intrus n'a aucune connaissance

Cette configuration initiale est représentée par l'automate de la figure 5.3 et est représentée par le langage :

$$State(nil, cc, tc, nil)$$

où

$$cc = ConverterCard([Key(old_{cw})^*, Key(current_{cw}), Key(next_{cw})^*], nil, nil)$$

et

$$tc = TerminalCard(nil, nil, VorI_0)$$

Nous effectuons la complétion de cet automate par  $\mathcal{R}_{proto}$  avec la fonction d'approximation qui essaie d'abord de normaliser les transitions à ajouter à l'aide des transitions existantes dans l'automate puis utilise les règles d'approximations (données dans la syntaxe *Timbuk* en annexe

B.2). Après 14 étapes de complétion, nous obtenons un automate complet contenant 210 transitions, cet automate est donné en annexe C.1. L'étude de cet automate permet de connaître l'ensemble des états atteints par les agents pour tout fonctionnement *normal*. Ainsi nous pouvons vérifier que le terminal card ne peut lire qu'une seule fois un même control-word. En effet un utilisant le même algorithme recherchant les instances d'un terme dans un langage (voir section 3.3.2), nous pouvons rechercher dans l'automate des sous-termes reconnus par l'automate de la forme

```
read(cw(x), read(cw(y), read(cw(z), u)))
```

Cette étude nous permet de dire qu'un même *Control Word* ne peut être reçu qu'au plus une fois, en effet il n'existe aucun 4-uplets solution  $(x, y, z, u)$  où l'état *qcurrent* qui ne reconnaît que le terme *current* apparaisse plus d'une fois. Par ailleurs, les solutions suivent l'ordre inverse de la liste des *Control Word* que le *Converter Card* doit envoyer, à savoir *next*, *current*, *old* ce qui implique que les *Control Word* sont reçus dans le bon ordre.

## 5.2 SmartRight et l'intrus

Nous disposons maintenant d'un automate de référence qui décrit le comportement normal du protocole, l'approximation qui a servi pour cette complétion sera utilisée comme point de départ pour les approximations suivantes : toutes les nouvelles transitions à ajouter seront forcément produites par les nouvelles règles. A partir de maintenant, nous allons pouvoir étudier les variations du comportement du protocole suivant la présence d'un intrus, suivant son pouvoir, son savoir. L'intrus est représenté par son savoir codé sous la forme d'une pile d'informations appelée *store*. Par exemple, si l'intrus connaît un message *m* ainsi qu'une clé *k1*, il sera représenté par le terme `store(m, store(k1, emptystore))`, le terme `emptystore` représentant l'information vide. Chaque transition de la forme `store(x, y)` sera normalisée vers l'état *qstore* ainsi chaque message *m* connu par l'intrus est représenté dans l'automate par une transition de la forme `store(m, qstore)`. Cette façon de normaliser permet de reconnaître le savoir de l'intrus par un ensemble de message non ordonné. Supposons que l'intrus ait connaissance de deux informations  $i_1$  et  $i_2$ , le sous automate dont les transitions sont

### Transitions

$i_1$	$\rightarrow$	$qi_1$
$i_2$	$\rightarrow$	$qi_2$
<code>store(<math>qi_1</math>, <math>qstore</math>)</code>	$\rightarrow$	$qstore$
<code>store(<math>qi_2</math>, <math>qstore</math>)</code>	$\rightarrow$	$qstore$
<code>emptystore</code>	$\rightarrow$	$qstore$

reconnait dans l'état *qstore* aussi bien la pile  $[i_1, i_2, \emptyset]$  que les piles  $[i_2, i_1, \emptyset]$ ,  $[\emptyset]$ ,  $[i_1, i_1, i_2, i_1, i_2]$ ,  $[i_1, i_1]$  ...

En particulier toute information peut se trouver en tête de pile.

### 5.2.1 Rejeu

Considérons tout d'abord un intrus dont les pouvoirs se limitent à

1. récupérer et stocker les messages qui circulent
2. renvoyer les messages qu'il connaît

Ces deux actions se modélisent aisément par des règles de réécriture sur les configurations :

1. la récupération des messages :

```
State(X, TC, CC, ST)
->
State(X, TC, CC, store(X,ST))
```

2. l'envoi de messages : cette seconde règle permet à l'intrus d'envoyer n'importe quel élément en tête de pile. Comme nous venons de montrer que n'importe quelle information peut se trouver en tête de pile, l'intrus peut renvoyer des messages qu'il a pu récupérer à l'aide de cette règle :

```
State(Y, TC, CC, store(X,ST))
->
State(X, TC, CC, store(X,ST))
```

Nous nous apercevons que la présence de l'intrus va modifier la liste des *Control Word* reçus par le *Terminal Card*, pour commencer les *Control Word* peuvent être perdus : si nous recherchons dans l'automate les instances possibles du terme

```
read(cw(x), read(cw(y), read(cw(z), u)))
```

nous trouvons par exemple les solutions

$$[x \mapsto gold, y \mapsto gold, z \mapsto gold, u \mapsto qnil]$$

$$[x \mapsto gold, y \mapsto gold, z \mapsto gold, u \mapsto qreadold]$$

Par ailleurs, ce nouvel automate contient en particulier les transitions utiles :

**Transitions**

```
read(qcurrent, qnil) → qreadcurrent
read(qcurrent, qreadcurrent) → qreadcurrent
```

Rappelons que notre approximation nous permet de distinguer de manière unique le *Control Word* courant `current_cw`, de même il existe une bijection entre l'état *qcurrent* et le terme `current_cw`, la première transition

$$read(qcurrent, qnil) \rightarrow qreadcurrent$$

signifie que le *Terminal Card* accepte une fois le *Control Word* étudié mais la seconde transition

$$read(qcurrent, qreadcurrent) \rightarrow qreadcurrent$$

indique que ce même *Control Word* peut être accepté une infinité de fois, ce qui est une faille du protocole. La question qui se pose maintenant est de savoir si nous avons mis en évidence une réelle faille du protocole ou si ces transitions ont été produites à cause d'un mauvais choix de la fonction d'abstraction. Notre première idée est de refaire la complétion de l'automate en utilisant une approximation exacte. Il est clair que le processus de complétion ne terminera sans doute pas, en revanche puisque les termes ajoutés seront exactement les descendants des termes initiaux. S'il existe une étape de complétion où l'automate reconnaît un sous terme de la forme

```
read(current_cw,read(current_cw,X))
```

pour n'importe quelle instance de  $X$  alors comme nous avons montré que chaque terme ajouté est issu d'un terme reconnu à l'étape précédente 3.4.2 alors nous aurons montré que le *Control Word* accepte au moins deux fois de suite le même *Control Word*. C'est bien le cas, nous avons montré qu'un même *Control Word* peut être reçu plusieurs fois.

Il reste maintenant à comprendre d'où vient la faille. Nous savons que sans la présence de l'intrus le protocole a le comportement attendu mais que si l'intrus peut rejouer les messages alors une erreur apparaît. Ceci est dû aux étapes 3-4 du protocole : il n'est pas spécifié que le *Terminal Card* change d'état une fois le message  $VoR||MAC(Vo_{key}, Vo_{Ri})$  reçu. Par défaut nous supposons que non, ainsi si l'intrus se contente de copier et renvoyer le message  $VoR||MAC(Vo_{key}, Vo_{Ri})$  alors le terminal card accepte une seconde fois d'extraire  $CW$  car dans la spécification initiale, les variables contenant les clés n'étaient réinitialisé qu'à la réception du premier message. Ceci n'est plus possible une fois que la session suivante commence et que le terminal card a reçu un nouveau control-word. Pour remédier à ceci, il suffit d'effacer la clé  $Vo_{Ri}$  et l'information courante une fois le test de comparaison réussi. Ce changement du protocole induit une modification de la modélisation :

nous replaçons la règle codant les étapes 3-4 décrite en 5.1.2 par

```
State(cons(VoR,cons(encrypt(VoKEY,Hach(VoRi)),nil)), CC,
      TerminalCard(cons(VoKEY, cons(Xor(CW,VoR), nil)), VoRi, CWlist), I)
->
State(nil, CC,
      TerminalCard(nil, nil, read(CW, CWlist)),I)
```

Une fois ce changement opéré, l'étude du nouvel automate permet d'affirmer que le *Converter Card* ne peut pas recevoir plus d'une fois un même *Control Word*. Nous avons signalé cette faille à *Thomson Multimédia* qui nous a affirmé qu'elle relevait en réalité d'une omission dans la spécification et que l'implémentation était correcte. La spécification a été mise à jour depuis mais cela met en évidence la difficulté de valider un système partant de la spécification. Dans l'idéal, une fois la spécification prouvée, il faudrait encore valider toutes les étapes menant à l'implémentation.

### 5.2.2 Un intrus capable de chiffrer, déchiffrer, composer et décomposer

Nous allons maintenant enrichir les pouvoirs de l'intrus afin de décrire un intrus suivant le modèle de Dolev Yao : supposons maintenant qu'il lui est aussi possible

1. de lire un message chiffré s'il possède la clé correspondante
2. de récupérer les différentes parties d'un message
3. de composer des messages chiffrés ou non à l'aide de ses informations.

Comme précédemment, nous allons représenter ces actions par des règles de réécriture :

```
store(encrypt(k,m),store(k,st))
->
store(m,store(encrypt(k,m),store(k,st)))
```

Cette première règle exprime le déchiffrement. Si le store de l'intrus contient un message  $m$  chiffré avec une clé  $k : \{m\}_k$  et s'il contient aussi cette même clé  $k$  alors on ajoute le corps  $m$  du message dans le store.

Exprimer qu'un intrus peut décomposer des messages, c'est à dire que si un message est formé de deux parties  $\{m_1, m_2\}$  alors il peut récupérer les informations  $m_1$  et  $m_2$ . Ceci s'exprime par la règle :

$\text{store}(\text{cons}(x,L), \text{st}) \rightarrow \text{store}(x, \text{store}(L, \text{store}(\text{cons}(x,L), \text{st})))$

La combinaison de ces deux nouvelles règles permet en particulier à l'intrus de récupérer l'ensemble des informations contenues dans un message chiffré lorsqu'il possède la clé.

A partir de l'automate complet, nous pouvons vérifier que le comportement du *Converter Card* n'a pas changé. Les deux premières propriétés sont vérifiées comme précédemment. Il nous reste maintenant à vérifier que l'intrus ne peut pas connaître les *Control Word*, la dernière propriété est vérifiée par défaut puisque nous n'avons pas encore modélisé un intrus émettant des *Control Word*. Pour savoir si l'intrus est capable de récupérer les *Control Word*, il suffit de rechercher dans l'automate complet les sous termes de la forme  $\text{store}(X,Y)$ , la propriété est vérifiée puisque aucun *Control Word* n'apparaît dans le store : aucun terme modélisant un *Control Word* n'est une instance de  $X$ .

### 5.2.3 Un intrus connaissant $\mathcal{K}$

Nous proposons maintenant d'étudier le comportement du protocole en présence d'un intrus connaissant la clé  $\mathcal{K}$ . Rappelons que cette clé est partagée par toutes les cartes de l'espace domestique, nous avons admis qu'aucune carte étrangère ne connaissait  $\mathcal{K}$ . Nous montrons maintenant que cette hypothèse est indispensable.

Modélisons un intrus connaissant cette clé : cette nouvelle hypothèse modifie le savoir initial de l'intrus mais pas ses actions, il suffit donc de modifier uniquement l'automate initial. La seule chose à modifier est le store initial : le savoir de l'intrus était jusqu'ici réduit à vide ( $\text{emptystore}$ ). Ajouter la clé  $\mathcal{K}$  revient simplement à ajouter les transitions suivantes :

**Transitions**

$k \rightarrow qk$   
 $\text{store}(qk, qstore) \rightarrow qstore$

Si les pouvoirs de l'intrus se limitent à ceux décrits dans la section précédente alors l'intrus ne peut pas récupérer le flux de *Control Word* puisqu'il n'apparaissent pas dans ses informations, ceci est dû au fait que les *Control Word* sont encore protégés par  $Xor$ . Nous allons encore enrichir les pouvoirs de l'intrus, nous permettons à l'intrus de décomposer  $\text{xor}(X,Y)$  lorsqu'il connaît au moins un élément qui le compose. Ce qui se code par les règles de réécriture suivantes :

$\text{store}(\text{xor}(X,Y), \text{store}(Y,ST)) \rightarrow \text{store}(X, \text{store}(\text{xor}(X,Y), \text{store}(Y,ST)))$   
 $\text{store}(\text{xor}(Y,X), \text{store}(Y,ST)) \rightarrow \text{store}(X, \text{store}(\text{xor}(X,Y), \text{store}(Y,ST)))$

Lors de la complétion de ce nouvel automate apparaissent les transitions

**Transitions**

$\text{store}(qcurrent, qstore) \rightarrow qstore$   
 $\text{store}(qnext, qstore) \rightarrow qstore$   
 $\text{store}(qold, qstore) \rightarrow qstore$

qui montrent que l'intrus a pu récupérer le flux de control-word. A partir de là, l'intrus est capable de jouer le rôle du *Converter Card*. Et donc le *Terminal Card* va accepter les *Control Word* dans n'importe quel ordre, une infinité de fois.

#### 5.2.4 Un intrus générant des *Control Word*

La dernière partie de notre étude consiste à étudier le comportement du protocole en présence d'un intrus capable de générer ses propres *Control Word*, ses propres clés (distinctes de  $\mathcal{K}$ ) et de composer des *Xor*, nous allons montrer que s'il ne connaît pas  $\mathcal{K}$  alors le terminal card n'accepte pas de *Control Word* malhonnêtes. Pour cela il faut encore modifier l'automate initial :

1. Nous ajoutons tout d'abord un nouveau *Control Word* dans le store, que nous nommons *spy\_cw* pour le différencier de ceux du *Converter Card* :

$$spy\_cw \rightarrow qcwspy$$

le flux est représenté par le langage  $spy\_cw, Succ(spy\_cw), \dots$  ce qui se code dans l'automate par la transitions suivante :

$$succ(qcwspy) \rightarrow qcwspy$$

2. de la même façon, nous représentons un flux de clés :

$$\begin{aligned} K\ spy &\rightarrow qkki \\ Key(qkki) &\rightarrow qki \\ store(qki, qstore1) &\rightarrow qstore1 \end{aligned}$$

Il reste la règle de composition des *Xor* :

$$store(x, store(y, st)) \rightarrow store(Xor(x, y), store(x, store(y, st)))$$

L'automate complet montre que le *Converter Card* n'accepte aucun *Control Word* généré par l'intrus car il n'existe pas de transitions de la forme

$$read(qcwspy, \star) \rightarrow \star$$

Ceci est du au fait que l'intrus ne connaît pas la clé  $\mathcal{K}$ . En revanche, si nous ajoutons la clé  $\mathcal{K}$  au savoir de l'intrus alors la propriété n'est plus vérifiée.

## 5.3 Conclusion

Nous avons présenté l'étude du protocole *View Only* de **SmartRight** à l'aide de l'outil *Timbuk*. Le premier travail a été l'étude du protocole sans intrus : le protocole est représenté par un ensemble de règles de réécriture agissant sur un automate d'arbre codant une situation initiale *normale*. La complétion de ce premier automate nous donne une sur-approximation de l'ensemble des configurations atteignables par le protocole à partir de la situation initiale décrite. Cette première étape nous a permis de vérifier que notre modélisation correspondait au déroulement du protocole, mais aussi de mettre en place une fonction d'approximation.

Nous avons ensuite étudié le comportement du protocole en présence d'un intrus capable simplement de rejouer, à n'importe quel instant, les messages qui circulent. Cette étude a permis de déceler une faille dans la spécification. La suite du travail a consisté à étudier le comportement

du protocole en présence d'un intrus capable de chiffrer/déchiffrer et composer/décomposer des messages. L'étude de l'automate obtenu ne montre pas de modification du comportement. Pour finir nous avons étudié le comportement du protocole en présence d'un intrus connaissant la clé  $\mathcal{K}$ , puis en présence d'un intrus capable de générer ses propres clés et ses propres  $\mathcal{K}$ .

L'ensemble de ce travail nous permet de conclure que la sécurité du protocole repose sur le partage de la clé  $\mathcal{K}$ . La seule connaissance de cette clé permet à l'intrus de récupérer le flux des *Control Word*. En revanche pour que le *Converter Card* accepte des *Control Word* erronés (soit déjà lus, soit produits par l'intrus), il faut en plus qu'il soit capable de générer des clés. Par ailleurs, nous avons montré que même si l'intrus génère son propre flux de *Control Word*, il ne peut pas les faire accepter au *converter card* s'il ne possède pas la clé  $\mathcal{K}$ . Nous avons montré que si la clé  $\mathcal{K}$  n'est pas divulguée alors le protocole vérifie les propriétés d'*Anti-Replay*, de secret et d'authentification telles que nous les avons définies en 5.1.1.

Troisième partie

Conjectures négatives



6

# Théories, modèles et preuves

## Sommaire

<b>6.1</b>	<b>Théories &amp; modèles</b> . . . . .	<b>72</b>
6.1.1	Langages, termes et formules . . . . .	72
6.1.2	Interprétation et satisfaction . . . . .	73
6.1.3	Modèles de Herbrand . . . . .	75
6.1.4	Théorie inductive . . . . .	76
<b>6.2</b>	<b>Méthodes de preuves</b> . . . . .	<b>78</b>
6.2.1	Récurrence explicite . . . . .	78
6.2.2	Preuve par cohérence . . . . .	78
6.2.3	Récurrence par réécriture . . . . .	82

A partir d'un système de réécriture  $\mathcal{R}$  et d'un ensemble régulier de termes  $E$  représentable par un automate  $\mathcal{A}$ , nous avons introduit la complétion d'automate, qui offre la possibilité de calculer un automate  $\mathcal{A}_{\mathcal{R},\alpha}^*$  reconnaissant un sur-ensemble des termes atteignables par  $\mathcal{R}$  à partir de  $\mathcal{A}$ . En outre, la complétion d'automate permet de montrer que des termes, ou des langages réguliers, sont inatteignables. Nous allons montrer comment cette technique permet de prouver de manière automatique des propriétés initiales sur le plus petit modèle de Herbrand d'une théorie  $\mathcal{E}$ . Pour cela, nous rappelons tout d'abord des notions basiques de théories des modèles, modèles de Herbrand, et théorèmes inductifs avant de présenter différentes méthodes de preuves.

## 6.1 Théories & modèles

### 6.1.1 Langages, termes et formules

**Définition 38 (Langages)** *Un langage du premier ordre est un ensemble  $\mathcal{L}$  formé de la réunion des sous ensembles suivants :*

- *Un ensemble infini dénombrable  $\{x_0, \dots, x_n, \dots\}$  de symboles de variables*
- *Un ensemble de symboles de connecteurs  $\{(\ , \ ), \neg, \wedge, \vee, \Rightarrow, \Leftrightarrow, \exists, \forall\}$*
- *$(\bigcup_{n \in \mathbb{N}} \mathcal{F}_n)$  où  $\mathcal{F}_n$  représente un ensemble de symboles de fonctions d'arité  $n$*
- *$(\bigcup_{n \in \mathbb{N}} \mathcal{P}_n)$  où  $\mathcal{P}_n$  représente un ensemble de symboles de prédicats d'arité  $n$  (notons que les deux derniers ensembles sont disjoints)*

*le symbole d'égalité  $\simeq$  est un symbole de  $\mathcal{P}_2$  particulier. Un langage dont le seul symbole de prédicat est  $\simeq$ , est appelé langage égalitaire.*

Notons que les symboles de constantes sont les éléments de  $\mathcal{F}_0$ .

**Exemple 13**  $\mathcal{L}_{ex}$  un petit langage

- $\mathcal{C} = \{a, b\}$
- $\mathcal{F}_2 = \{p, m\}$
- $\mathcal{P}_2 = \{\simeq\}$

**Définition 39 (Formules)** *Pour un langage  $\mathcal{L}$  donné, une formule atomique  $\mathcal{F}$  est un mot de  $\mathcal{L}$  de la forme  $\mathcal{P}(t_1, \dots, t_n)$  où  $\mathcal{P} \in \mathcal{P}_n$  et  $t_1, \dots, t_n$  sont des termes. L'ensemble des formules de  $\mathcal{L}$  est défini par induction :*

- *Les formules atomiques sont des formules*

- si  $\mathcal{F}_1$  et  $\mathcal{F}_2$  sont des formules et  $x$  une variable alors  $\neg\mathcal{F}_1$ ,  $\mathcal{F}_1 \vee \mathcal{F}_2$ ,  $\mathcal{F}_1 \wedge \mathcal{F}_2$ ,  $\mathcal{F}_1 \Rightarrow \mathcal{F}_2$ ,  $\mathcal{F}_1 \Leftrightarrow \mathcal{F}_2$ ,  $\exists x\mathcal{F}_1$  et  $\forall x\mathcal{F}_1$  sont des formules.

En particulier, si  $\mathcal{L}$  est égalitaire les formules atomiques de  $\mathcal{L}$  sont les  $t_1 \simeq t_2$  pour tous couples de termes  $(t_1, t_2)$ .

**Définition 40 (variables libres)** *Intuitivement les variables libres sont les variables qui ne dépendent pas d'un quantificateur, plus formellement :*

- toutes les variables d'une formule atomique sont libres
- si  $\mathcal{F}$  est une formule de la forme  $\neg\mathcal{F}_1$ ,  $\mathcal{F}_1 \vee \mathcal{F}_2$ ,  $\mathcal{F}_1 \wedge \mathcal{F}_2$ ,  $\mathcal{F}_1 \Rightarrow \mathcal{F}_2$ ,  $\mathcal{F}_1 \Leftrightarrow \mathcal{F}_2$ , alors les variables libres de  $\mathcal{F}$  sont les variables libres de  $\mathcal{F}_1$  et  $\mathcal{F}_2$
- si  $\mathcal{F}$  est de la forme  $\exists x\mathcal{F}_1$  ou  $\forall x\mathcal{F}_1$  et si  $x$  apparaît dans  $\mathcal{F}_1$  alors  $x$  n'est pas libre dans  $\mathcal{F}$ ,  $x$  est dite liée dans  $\mathcal{F}$ .

Une formule close est une formule qui n'a pas de variable libre. Si  $\mathcal{F}$  est une formule dont les seules variables libres sont  $x$  et  $y$  alors  $\forall x \forall y \mathcal{F}$  est une formule close appelée *cloture universelle* de  $\mathcal{F}$ . Une théorie sur un langage  $\mathcal{L}$  est un ensemble de formules closes sur  $\mathcal{L}$ .

### 6.1.2 Interprétation et satisfaction

#### Interprétation d'un langage

**Définition 41 (Interprétation d'un langage)**

*Soit  $\mathcal{L}$  un langage, une interprétation de  $\mathcal{L}$  est une structure  $\mathfrak{M}$  contenant un ensemble de base  $M$  tel que*

- pour tout symbole de constante  $c$  de  $\mathcal{L}$  il existe un élément  $\bar{c} \in M$  appelé interprétation de  $c$
- pour tout symbole de fonction  $f$  d'arité  $n$  de  $\mathcal{L}$  il existe une fonction  $\bar{f}$  de  $M^n \rightarrow M$  appelée interprétation de  $f$
- pour tout symbole de prédicat  $P$  d'arité  $n$  de  $\mathcal{L}$  il existe une relation  $n$ -aire  $\bar{P}$  sur  $M$  appelée interprétation de  $P$

**Exemple 14**  $\langle \mathbb{Z}, 0, 1, +, \times \rangle$  est une interprétation de  $\mathcal{L}_{ex}$

- L'ensemble de base est l'ensemble des entiers relatifs :  
 $\mathbb{Z}$
- Les constantes :  
 $\bar{a} = 0$   
 $\bar{b} = 1$
- Les symboles de fonction  
 $\bar{p} = +$   
 $\bar{m} = \times$
- Le symbole de prédicat  
 $\bar{\simeq} = \{(x, x) \mid x \in \mathbb{Z}\}$  et donc  $\bar{\simeq}$  est l'égalité

de même  $\langle \mathbb{R}, 0, 1, +, \times \rangle$  est une interprétation de  $\mathcal{L}_{ex}$

#### Satisfaction d'une formule

**Définition 42 (Satisfaction d'une formule dans une interprétation)**

*Considérons*

$\mathcal{F} = \mathcal{F}[x_1, \dots, x_n]$  une formule de  $\mathcal{L}$ ,

$\mathfrak{M}$  une interprétation de  $\mathcal{L}$ ,

$a_1, \dots, a_n$   $n$  éléments de  $\mathfrak{M}$ .

On dit que  $\mathfrak{M}$  satisfait  $\mathcal{F}$  de  $a_1, \dots, a_n$  ou que  $\mathfrak{M}$  est modèle de  $\mathcal{F}$  et on note  $\mathfrak{M} \models \mathcal{F}[a_1, \dots, a_n]$  si

- $\vee \mathcal{F} = P(t_1, \dots, t_k)$  et  $(\bar{t}_1[x_1 \leftarrow a_1, \dots, x_n \leftarrow a_n], \dots, \bar{t}_k[x_1 \leftarrow a_1, \dots, x_n \leftarrow a_n]) \in \bar{P}$
- $\vee \mathcal{F} = \neg G$  et  $\mathfrak{M} \not\models G$
- $\vee \mathcal{F} = \mathcal{F}_1 \wedge \mathcal{F}_2$  et  $\mathfrak{M} \models \mathcal{F}_1$  et  $\mathfrak{M} \models \mathcal{F}_2$
- $\vee \mathcal{F} = \mathcal{F}_1 \vee \mathcal{F}_2$  et  $\mathfrak{M} \models \mathcal{F}_1$  ou  $\mathfrak{M} \models \mathcal{F}_2$
- $\vee \mathcal{F} = \mathcal{F}_1 \Rightarrow \mathcal{F}_2$  et  $\mathfrak{M} \not\models \mathcal{F}_1$  ou  $\mathfrak{M} \models \mathcal{F}_2$
- $\vee \mathcal{F} = \mathcal{F}_1 \Leftrightarrow \mathcal{F}_2$  et  $(\mathfrak{M} \models \mathcal{F}_1$  et  $\mathfrak{M} \models \mathcal{F}_2)$  ou  $(\mathfrak{M} \not\models \mathcal{F}_1$  et  $\mathfrak{M} \not\models \mathcal{F}_2)$
- $\vee \mathcal{F} = \forall x \mathcal{F}_1$  avec  $x \notin \{x_1, \dots, x_n\}$  et pour tout élément  $a$  de  $M$  on a  $\mathfrak{M} \models \mathcal{F}_1[a, a_1, \dots, a_n]$
- $\vee \mathcal{F} = \exists x \mathcal{F}_1$  avec  $x \notin \{x_1, \dots, x_n\}$  et il existe un élément  $a$  de  $M$  vérifiant  $\mathfrak{M} \models \mathcal{F}_1[a, a_1, \dots, a_n]$
- $\vee \mathcal{F} = \forall x_i \mathcal{F}_1$  et pour tout élément  $a$  de  $M$  on a  $\mathfrak{M} \models \mathcal{F}_1[a_1, a_{i-1}, a, a_{i+1}, \dots, a_n]$
- $\vee \mathcal{F} = \exists x_i \mathcal{F}_1$  et il existe un élément  $a$  de  $M$  vérifiant  $\mathfrak{M} \models \mathcal{F}_1[a_1, a_{i-1}, a, a_{i+1}, \dots, a_n]$

Une formule est dite **satisfaisable** s'il existe une interprétation de  $\mathcal{L}$  dans laquelle elle est vraie, et une théorie (ensemble de formules) est dite consistante si elle admet au moins un modèle.

**Exemple 15** Pour notre exemple  $\mathcal{L}_{ex}$ , considérons la théorie  $\mathbf{T}$  formée des formules :

$$\mathcal{F}_1 = \exists y \forall x \simeq (p(x, y), x) \wedge \mathcal{F}_2 = \forall x \exists y \simeq (m(x, y), b)$$

Pour  $\langle \mathbb{R}, 0, 1, +, \times \rangle$  l'interprétation de  $\mathbf{T}$  donne

$$\overline{\mathcal{F}_1} = \exists y \forall x ((x + y) = x)$$

$$\overline{\mathcal{F}_2} = \forall x \exists y ((x \times y) = 1)$$

qui sont des formules **vraies** dans  $\langle \mathbb{R}, 0, 1, +, \times \rangle$ . L'interprétation  $\langle \mathbb{R}, 0, 1, +, \times \rangle$  est **modèle** de  $\mathbf{T}$ .

Pour  $\langle \mathbb{Z}, 0, 1, +, \times \rangle$  l'interprétation de  $\mathbf{T}$  est la même, mais  $\overline{\mathcal{F}_2}$  est **n'est pas vraie** dans  $\langle \mathbb{Z}, 0, 1, +, \times \rangle$  : dans  $\mathbb{Z}$  tout élément n'admet pas nécessairement d'inverse,  $\langle \mathbb{Z}, 0, 1, +, \times \rangle$  **n'est pas modèle** de  $\mathbf{T}$ .

Si  $\mathbf{T}$  est une théorie sur un langage  $\mathcal{L}$ , une interprétation  $\mathfrak{M}$  de  $\mathcal{L}$  est modèle de  $\mathbf{T}$  si et seulement si  $\mathfrak{M}$  est modèle de toutes les formules de  $\mathbf{T}$ . Si  $\mathbf{T}$  est une théorie et  $F$  une formule close de  $\mathcal{L}$  alors  $F$  est *conséquence sémantique* de  $\mathbf{T}$  si et seulement si tout modèle de  $\mathbf{T}$  est encore modèle de  $F$ . Par ailleurs, nous avons défini une théorie de  $\mathcal{L}$  comme un ensemble de formules closes de  $\mathcal{L}$ , mais plus précisément, une théorie est un ensemble de formules qui contient toutes ses conséquences. Si  $E$  est un ensemble consistant de formules, l'ensemble des conséquences de  $E$  est une théorie  $\mathbf{T}_E$  appelé théorie engendrée par  $E$ . Par la suite nous confondrons  $E$  et  $\mathbf{T}_E$ .

Arrêtons nous un instant pour bien comprendre la notion de modèle.

Les modèles sont les objets concrets qui interprètent certaines idées abstraites. Si nous savons,

par exemple, écrire sous forme de formules qu'une loi binaire sur un ensemble définit un groupe, alors nous savons écrire "la théorie des groupes". Un modèle de cette théorie est une structure qui vérifie tous les axiomes de cette théorie, c'est tout simplement un groupe. la notion de modèle suppose que l'on a interprété les symboles figurant dans les formules, et donc que l'on a donné un "sens" à ces formules, dire si ces formules sont *vraies* ou *fausses* dans ces structures est une notion *sémantique*. En revanche, une formule *démontrable* est une formule qui est vraie dans *tous* les modèles d'une théorie est dite *valide* ou *universellement valide*, une démonstration est une notion *syntactique*. Une démonstration est un raisonnement déductif qui permet d'établir la validité d'une formule à partir d'autres formules considérées comme vraies. Une démonstration ne fait pas intervenir la notion de modèle et donc clairement s'il existe une démonstration d'une formule  $\mathcal{F}$  dans une théorie  $\mathbf{T}$  alors  $\mathcal{F}$  est vraie dans tous les modèles de  $\mathbf{T}$ . S'il existe une démonstration de  $\mathcal{F}$  dans  $\mathbf{T}$ , nous dirons que  $\mathcal{F}$  est un théorème ou que  $\mathcal{F}$  est démontrable dans  $\mathbf{T}$  et ce qui se notera  $\mathbf{T} \vdash \mathcal{F}$ .

### 6.1.3 Modèles de Herbrand

Pour qu'une formule close  $\mathcal{F}$  d'un langage  $\mathcal{L}$  soit conséquence d'une théorie  $\mathcal{T}$ , il faut qu'elle soit vraie dans tous les modèles de  $\mathbf{T}$ , ce qui est impossible à vérifier en pratique. Nous présentons maintenant des structures particulières dites "libres", "syntactiques" ou "de Herbrand" puis nous étudierons les formules qui sont vraies dans les modèles de Herbrand.

#### Interprétation de Herbrand

##### Définition 43 (Univers de Herbrand)

Soit  $\mathcal{L}$  un langage, l'univers de Herbrand, que nous noterons  $\mathcal{G}$ , associé à  $\mathcal{L}$  est l'ensemble défini inductivement par

- si  $c$  est un élément de l'ensemble  $\mathcal{C}$  des symboles de constante de  $\mathcal{L}$  alors  $c \in \mathcal{G}$
- pour tout symbole de fonctions  $f$  d'arité  $n$  de  $(\bigcup_{n \in \mathbb{N}} F_n)$  de  $\mathcal{L}$ , et pour tout  $n$ -uplet  $t_1, \dots, t_n$  d'éléments de  $\mathcal{G}$ ,  $f(t_1, \dots, t_n) \in \mathcal{G}$

L'univers de Herbrand d'un langage  $\mathcal{L}$  est simplement l'ensemble des termes que l'on peut former à l'aide des symboles de constantes et des symboles de fonctions.

##### Définition 44 (Base de Herbrand)

Soit  $\mathcal{L}$  un langage, la base de Herbrand de  $\mathcal{L}$  est l'ensemble  $\mathcal{B}_{\mathcal{H}}$  des formules atomiques closes, ie l'ensemble des  $P(t_1, \dots, t_n)$  pour tout prédicat  $P$  de  $\mathcal{L}$  et tout  $n$ -uplet d'éléments  $t_1, \dots, t_n$  de  $\mathcal{G}$ .

##### Définition 45 (Structure de Herbrand)

Un même langage  $\mathcal{L}$  définit un unique univers de Herbrand, mais sur cet univers il est possible de définir plusieurs structures de Herbrand. Une structure de Herbrand  $\mathcal{H}$  est décrite par une **interprétation de Herbrand** qui est un sous ensemble  $I$  de la base de Herbrand,  $I$  définit les formules atomiques qui sont vraies dans la structure. Pour une base de Herbrand  $\mathcal{B}_{\mathcal{H}}$  donnée, il y a donc  $2^{\mathcal{B}_{\mathcal{H}}}$  interprétations de Herbrand.

**Exemple 16** Soit  $\mathcal{L} = \{a, f, \mathcal{P}, \mathcal{R}\}$  où  $a$  est un symbole de constante,  $f$  un symbole de fonction unaire,  $\mathcal{P}$  un symbole de prédicat unaire, et  $\mathcal{R}$  un symbole de prédicat binaire. L'univers de Herbrand est  $\{a, f(a), f(f(a)), \dots, f^n(a), \dots\}$ . La base de Herbrand est l'ensemble  $\mathcal{B}_{\mathcal{H}} = \{\mathcal{P}(t), \mathcal{R}(t, t') \mid \forall t, t' \in \mathcal{G}\}$ . Les ensembles suivants définissent des interprétations de Herbrand :

- $I_0 = \emptyset$ ,
- $I_1 = \{\mathcal{P}(a), \mathcal{P}(f(f(a)))\}$ ,
- $I_2 = \{\mathcal{P}(f(a)), \mathcal{R}(a, f^n(a)) \mid \forall n \in \mathbb{N}\}$ .

**Définition 46 (Modèle de Herbrand)**

Soit  $\mathbf{T}$  une théorie sur un langage  $\mathcal{L}$  et  $\mathcal{H}$  une structure de Herbrand pour  $\mathcal{H}$ , on dit que  $\mathcal{H}$  est un **modèle de Herbrand** de  $\mathbf{T}$  si et seulement si  $\mathcal{H}$  est modèle de  $\mathbf{T}$ .

**Définition 47 (plus petit modèle de Herbrand)**

Soit  $\mathcal{L}$  un langage,  $\mathcal{B}_{\mathcal{H}}$  sa base de Herbrand, et  $\mathcal{F}$  une formule sur  $\mathcal{L}$ , le plus petit modèle de Herbrand de  $\mathcal{F}$  sur  $\mathcal{L}$  est défini par

$$\bigcap_{\mathfrak{M} \in 2^{\mathcal{B}_{\mathcal{H}}}, \mathfrak{M} \models \mathcal{F}} \mathfrak{M}$$

**Exemple 17** Soit  $\mathcal{L} = \{a, f, \mathcal{P}, \mathcal{R}\}$  où  $a$  est un symbole de constante,  $f$  un symbole de fonction unaire,  $\mathcal{P}$  un symbole de prédicat unaire, et  $\mathcal{R}$  un symbole de prédicat binaire, et soit  $\mathbf{T} = \{\mathcal{P}(a)\}$ . Dans notre exemple précédent, seul  $I_2$  définit une structure de Herbrand qui soit modèle de  $\mathbf{T}$ .

### 6.1.4 Théorie inductive

Nous nous concentrons maintenant sur les théories dont le langage contient le prédicat d'égalité  $\simeq$ , elles sont appelées théories égalitaires. Dans ce cadre, nous rappelons qu'un littéral positif est soit une formule atomique (def 39), soit une équation de la forme  $t_1 \simeq t_2$  où  $t_1$  et  $t_2$  sont des termes. Un littéral est soit un littéral positif, soit sa négation, une clause est une disjonction de littéraux et en particulier, une clause de Horn est une clause qui contient exactement un littéral positif.

**Définition 48 (Théorème inductif)** Soit  $\mathbf{T}$  une théorie,  $\mathcal{F}$  une formule,  $\mathcal{F}$  est une conséquence inductive ou théorème inductif de  $\mathbf{T}$  si et seulement si pour toute interprétation de Herbrand  $\mathcal{H}$ ,  $\mathcal{H} \models \mathbf{T}$  implique  $\mathcal{H} \models \mathcal{F}$ .

L'ensemble des conséquences inductives d'une théorie  $\mathbf{T}$  forme la théorie inductive de  $\mathbf{T}$ .

**Définition 49 (Théorie équationnelle)** Pour un ensemble d'équations  $\mathcal{E}$  et un ensemble de termes  $\mathcal{T}(\mathcal{F}, \mathcal{X})$ , la théorie équationnelle de  $\mathcal{E}$  est l'ensemble des égalités obtenues à partir de  $\mathcal{E}$  et des règles suivantes :

(Les signatures sortées ont été définies en déf 1)

1. *Reflexivité*

$$\vdash (u \simeq u)$$

2. *Symétrie*

$$(u \simeq v) \vdash (v \simeq u)$$

3. *Transitivité*

$$(t \simeq u), (u \simeq v) \vdash (t \simeq v)$$

4. *Congruence*

$$\forall f : \mathcal{S}_1 \times \dots \times \mathcal{S}_n \rightarrow \mathcal{S}_{n+1} \text{ et } \forall u_1 \in \mathcal{S}_1, \dots, u_n \in \mathcal{S}_n, v_1 \in \text{Sort}_1, \dots, v_n \in \mathcal{S}_n, \\ (u_1 \simeq v_1), \dots, (u_n \simeq v_n) \vdash (f(u_1, \dots, u_n) \simeq f(v_1, \dots, v_n))$$

## 5. Substitution

$(u \simeq v) \vdash (u\sigma \simeq v\sigma)$  pour toute substitution  $\sigma$

Un modèle de Herbrand d'un système équationnel  $\mathcal{E} = \{u_1 = v_1, \dots, u_n = v_n\}$  est un modèle  $\mathfrak{M}$  ayant pour ensemble de base les termes de  $\mathcal{T}(\mathcal{F}, \mathcal{X})$ , tel  $\mathfrak{M}$  satisfait la formule  $u_1 = v_1 \vee \dots \vee u_n = v_n$

Une théorie équationnelle  $\mathbf{T}$  engendre une relation de congruence notée  $\equiv_{\mathbf{T}}$  sur les termes  $\mathcal{T}(\mathcal{F}, \mathcal{X})$ , définie comme la plus petite relation de congruence sur  $\mathcal{T}(\mathcal{F}, \mathcal{X})$  telle que  $l\sigma \equiv_{\mathbf{T}} r\sigma$  pour toute équation  $l = r$  de  $\mathbf{T}$ . Le plus petit modèle de Herbrand ou modèle *initial* d'une théorie équationnelle  $\mathbf{T}$  est simplement l'algèbre quotient  $\mathcal{T}(\mathcal{F}) / \equiv_{\mathbf{T}}$  et dans ce cas particulier, une équation est un théorème inductif si et seulement si elle est vraie dans  $\mathcal{T}(\mathcal{F}) / \equiv_{\mathbf{T}}$ .

## Théorie équationnelle et réécriture

Soit  $\mathcal{R} = \{l_i \rightarrow r_i\}_{i \in I}$  un système de réécriture,  $\mathcal{R}$  induit un ensemble d'équations simplement défini par  $\mathbf{T} = \{l_i = r_i\}_{i \in I}$ . Ces deux ensembles génèrent une même relation de congruence sur un ensemble de termes :

**THEOREME 5: Birkhoff**

Soit  $\mathcal{R} = \{l_i \rightarrow r_i\}_{i \in I}$  un système de réécriture et  $\mathbf{T} = \{l_i = r_i\}$  la théorie équationnelle associée, pour tous termes  $u$  et  $v$ , on a  $u =_{\mathbf{T}} v$  si et seulement si il existe une séquence finie de termes  $t_0, \dots, t_n$  telle que  $t_0 = u$ ,  $t_n = v$  et pour tout  $1 \leq i \leq n-1$  on a soit  $t_i \rightarrow_{\mathcal{R}} t_{i+1}$ , soit  $t_i \leftarrow_{\mathcal{R}} t_{i+1}$ .

A partir de maintenant, nous parlerons aussi bien de  $\mathcal{R}$  que de la théorie  $\mathbf{T}$  associée. Jusqu'ici nous avons utilisé la réécriture pour montrer que des termes sont inatteignables à partir d'un ensemble régulier de termes et d'un système de réécriture donné. Nous allons étudier la *portabilité* de nos algorithmes pour faire des preuves négatives dans le plus petit modèle de Herbrand d'une théorie, c'est à dire que nous souhaitons prouver des propriétés de la forme  $t_1 \neq t_2$  dans  $\mathcal{T}(\mathcal{F}) / \equiv_{\mathbf{T}}$ , ce que nous nommerons propriétés initiales. C'est un problème distinct de celui des théorèmes inductifs, les théorèmes inductifs sont des propriétés vraies dans tous les modèles de  $\mathbf{T}$  alors que les propriétés initiales ne sont en général vraies que dans  $\mathcal{T}(\mathcal{F}) / \equiv_{\mathbf{T}}$ . La plupart des méthodes de démonstration automatique s'intéressent à prouver des équations dans des théories purement équationnelles<sup>12</sup>. En effet, si  $\mathbf{T}$  est une théorie purement équationnelle alors elle définit une relation d'équivalence sur les termes qui induit l'algèbre quotient  $\mathcal{T}(\mathcal{F}) / \equiv_{\mathbf{T}}$  qui est son plus petit modèle de Herbrand. Supposons que dans cette théorie, nous essayions de prouver une formule de la forme  $\mathcal{F} = \neg(t_1 = t_2)$ <sup>13</sup> ce que nous appellerons une diséquation. Si  $\mathcal{F}$  était démontrable alors,  $\mathcal{F}$  est vraie dans tout les modèles de  $\mathbf{T}$ , ce qui est impossible. En effet, nous pouvons toujours construire un modèle  $\mathfrak{M}$  de  $\mathbf{T} \cup \neg\mathcal{F}$  puisque c'est à nouveau une théorie équationnelle,  $\mathfrak{M}$  est un modèle de  $\mathbf{T}$  qui n'est pas modèle de  $\mathcal{F}$ .

**Exemple 18** *Considérons par exemple la théorie  $\mathbf{T}$  sur le langage  $\{0 : 0, Succ : 1, even : 1, odd : 1, True : 0, False : 0\}$  et défini par :*

<sup>12</sup>en réalité le principe s'étend aux théories qui sont des ensemble de clauses.

<sup>13</sup>plus formellement nous supposons  $t_1, t_2 \in \mathcal{T}(\mathcal{F}, \mathcal{X})$  avec  $Var(t_1) = \{x_1, \dots, x_n\}$  et  $Var(t_2) = \{y_1, \dots, y_n\}$   $\neg(t_1 = t_2)$  ou  $(t_1 \neq t_2)$  est un abus de langage pour désigner la formule  $\forall x_1 \dots x_n, \forall y_1, \dots, y_n \neg(t_1 = t_2)$

$$\text{even}(0) = \text{True}$$

$$\text{odd}(0) = \text{False}$$

$$\text{even}(\text{Succ}(x)) = \text{odd}(x)$$

$$\text{odd}(\text{Succ}(0)) = \text{even}(x)$$

Le plus petit modèle de Herbrand de cette théorie est  $\mathcal{T}(\mathcal{F})/\equiv_{\mathbf{T}}$  dans laquelle  $\text{True}$  est distinct de  $\text{False}$ . Dans  $\mathcal{T}(\mathcal{F})/\equiv_{\mathbf{T}}$ , nous considérons la formule  $\mathcal{F} = \forall x \neg(\text{even}(x) = \text{odd}(x))$ . Sans rentrer dans les détails pour l'instant, ceci se montre simplement en raisonnant par cas sur  $x$ . En revanche, toute interprétation de Herbrand qui soit modèle de  $\mathcal{T}$  et dans laquelle la formule  $\text{True} = \text{False}$  est vérifiée, n'est pas modèle de  $\mathcal{F}$ . Notre formule  $\mathcal{F}$  est vraie dans  $\mathcal{T}(\mathcal{F})/\equiv_{\mathbf{T}}$  mais ne l'est pas dans tous les modèles de Herbrand de  $\mathbf{T}$ , c'est une propriété initiale qui n'est pas un théorème inductif.

Avant de proposer un système de preuve pour les propriétés initiales, nous allons étudier les diverses approches déjà développées pour la démonstration assistée ou automatique.

## 6.2 Méthodes de preuves

### 6.2.1 Récurrence explicite

La récurrence explicite s'appuie sur un ordre bien fondé sur les éléments de  $\mathcal{T}(\mathcal{F})/\equiv_{\mathbf{T}}$ , la preuve d'une formule  $\mathcal{F}$  se déroule en plusieurs *étapes de récurrence*, qui comprend une instance de  $\mathcal{F}$  à prouver à l'aide éventuellement d'une ou plusieurs hypothèses de récurrence qui doivent être plus petite selon l'ordre que  $\mathcal{F}$ . En général, l'ordre utilisé est celui induit par l'ordre sous termes, c'est la récurrence structurelle. Par exemple, sur les entiers, la récurrence sur les entiers est la suivant :

*Si il est possible de montrer qu'une propriété  $\mathcal{F}$  est vraie pour 0 et que par ailleurs pour tout nombre  $x$  qui la satisfait,  $\text{Succ}(x)$  satisfait encore  $\mathcal{F}$ , alors on a montré  $\mathcal{F}$  pour tous les nombres.*

La récurrence explicite est implantée dans les assistants à la preuve comme RRL [KZ95], PVS [ORR+96] ou COQ [BBC+99]. Dans COQ, "les raisonnements sur les fonctions structurelles se font par récurrence sur l'argument principal puis en suivant la structure des règles de définitions (filtrage) de ces fonctions" [BC03]. Le principal avantage de la récurrence explicite est qu'elle génère des preuves faciles à comprendre malheureusement ces preuves restent difficiles à automatiser.

### 6.2.2 Preuve par cohérence

La preuve dite par cohérence n'utilise pas explicitement de principe de récurrence. Intuitivement, supposons que nous souhaitions démontrer par cohérence une formule  $\mathcal{F}$ , nous supposons  $\neg\mathcal{F}$  et tâcherions d'aboutir à une contradiction. Plus formellement, si l'on considère une théorie équationnelle  $\mathbf{T}$ , le principe de la preuve par cohérence est d'abord de trouver une axiomatisation de  $\mathcal{T}(\mathcal{F})/\equiv_{\mathbf{T}}$  :

**Définition 50 (Axiomatisation)** Soit  $\mathbf{T}$  une théorie équationnelle. Une axiomatisation de  $\mathcal{T}(\mathcal{F})/\equiv_{\mathbf{T}}$  est un ensemble de formules *Axiom* universellement quantifiées qui caractérise  $\mathcal{T}(\mathcal{F})/\equiv_{\mathbf{T}}$ , plus précisément :

- $\mathcal{T}(\mathcal{F})/\equiv_{\mathbf{T}}$  est modèle de *Axiom*
- si  $\mathfrak{M}$  est un modèle de  $\mathbf{T} \cup \text{Axiom}$  alors  $\mathfrak{M}$  est isomorphe à  $\mathcal{T}(\mathcal{F})/\equiv_{\mathbf{T}}$

Ensuite il s'agit d'essayer de dériver une contradiction entre la formule  $\mathcal{F}$  à prouver et *Axiom*. En effet, s'il est possible de mettre en évidence une contradiction entre  $\mathcal{F}$  et *Axiom*, alors le plus petit modèle de  $\mathbf{T} \cup \mathcal{A}$  est distinct de  $\mathcal{T}(\mathcal{F})/\equiv_{\mathbf{T}}$  par définition de *Axiom*, donc  $\mathcal{F}$  n'est pas une formule vraie dans  $\mathcal{T}(\mathcal{F})/\equiv_{\mathbf{T}}$ , ce n'est donc pas un théorème inductif. Nous proposons d'illustrer ce principe par un exemple repris de [Com94] :

**Exemple 19** Considérons la conjecture  $\mathcal{F} = \text{Succ}(x)+0 = \text{Succ}(0)$  dans la théorie  $\mathbf{T}$  suivante :

$$0 + x = x \quad (6.1)$$

$$\text{Succ}(x) + y = \text{Succ}(x + y) \quad (6.2)$$

$$(6.3)$$

Le plus petit modèle de  $\mathbf{T}$  est donc l'algèbre quotient  $\mathcal{T}(\mathcal{F})/\equiv_{\mathbf{T}}$  que l'on caractérise par les formules suivantes (la quantification est implicitement universelle :

$$0 \neq \text{Succ}(x) \quad (6.4)$$

$$\text{Succ}(x) = \text{Succ}(y) \Rightarrow x = y \quad (6.5)$$

$$(6.6)$$

Pour trouver une contradiction, il suffit ici de réécrire le terme  $\text{Succ}(0)$  en utilisant la conjecture ainsi que les éléments de  $\mathbf{T}$  et de *Axiom* :

$$\text{Succ}(0) \xrightarrow{\mathcal{F}} \text{Succ}(x) + 0 \xrightarrow{\mathbf{T}} \text{Succ}(x + 0) \xrightarrow{\mathbf{T}} \text{Succ}(x)$$

Finalement nous avons montré que les termes  $\text{Succ}(0)$  et  $\text{Succ}(x)$  étaient équivalents suivant la relation  $\equiv_{\mathbf{T} \cup \text{Axiom} \cup \mathcal{F}}$  ce qui contredit le premier élément de l'axiomatisation.

Le principal problème posé par cette méthode est de trouver l'axiomatisation de  $\mathcal{T}(\mathcal{F})/\equiv_{\mathbf{T}}$  qui convient à la conjecture considérée, puisqu'une axiomatisation n'est pas nécessairement complète. Le second problème est de savoir prouver l'inconsistance de  $\mathbf{T} \cup \text{Axiom} \cup \mathcal{F}$ . Nous nous appuyerons sur [Com94] pour présenter maintenant les diverses approches de preuves par cohérence : la définition d'un prédicat *eq* par D. Musser [Mus80], des propriétés sur les constructeurs (G. Huet et J.M. Hullot [HH82]), et enfin la réductibilité inductive proposée par J.P. Jouannaud et E. Kounalis [JK89].

### L'approche de D. Musser

D. Musser considère une théorie  $\mathbf{T}$  contenant une fonction *eq* [Mus80] complètement définie<sup>14</sup> sur les termes du langage et de deux symboles de constantes *True* et *False*. *eq* caractérise

<sup>14</sup>voir ultérieurement la définition de la suffisante complétude

l'égalité modulo  $\mathbf{T}$ , en effet on supposera d'une part que le système de réécriture est convergent et que pour tous couples de termes clos  $(s, t)$  nous avons :

$$s \equiv_{\mathbf{T}} t \leftrightarrow eq(s, t) =_{\mathbf{T}} True \quad (6.7)$$

$$s \not\equiv_{\mathbf{T}} t \leftrightarrow eq(s, t) =_{\mathbf{T}} False \quad (6.8)$$

L'axiomatisation  $\mathcal{Axiom}$  se réduit alors à l'équation  $True \neq False$  :

1.  $\mathcal{T}(\mathcal{F}) / \equiv_{\mathbf{T}}$  est bien modèle de  $\mathcal{A}$  puisque  $True$  distinct de  $False$
2. et si  $\mathfrak{M}$  est un modèle de Herbrand de  $\mathbf{T} \cup \mathcal{Axiom}$  alors pour tout couple de termes clos  $(s, t)$  soit
  - $\mathfrak{M} \models \{s = t\}$  et  $s \equiv_{\mathbf{T}} t$
  - $\mathfrak{M} \not\models \{s = t\}$  et  $s \not\equiv_{\mathbf{T}} t$
  - $\mathfrak{M} \not\models \{s = t\}$  et  $s \equiv_{\mathbf{T}} t$  et alors  $\mathfrak{M}$  n'est pas modèle de  $\mathbf{T}$
  - $\mathfrak{M} \models \{s = t\}$  et  $s \not\equiv_{\mathbf{T}} t$  et alors par définition de  $eq$  on a  $True = eq(s, s) = eq(s, t) = False$  donc  $\mathfrak{M}$  n'est pas modèle de  $\mathcal{A}$
 donc finalement si  $\mathfrak{M}$  est modèle de  $\mathbf{T} \cup \mathcal{Axiom}$  alors pour tout couple de termes clos  $(s, t)$ ,  $s = t$  si et seulement si  $s \equiv_{\mathbf{T}} t$  donc  $\mathfrak{M} \approx \mathcal{T}(\mathcal{F}) / \equiv_{\mathbf{T}}$

A partir de là, le principe est de compléter le système  $\mathbf{T} \cup \mathcal{F}$ <sup>15</sup> en utilisant l'algorithme de Knuth Bendix [KB83]. L'inconsistance est détectée lorsque la complétion produit l'équation  $True = False$ . Nous proposons un exemple réalisé avec CiME [CM96], système qui permet entre autres d'effectuer la complétion d'un système de réécriture :

**Exemple 20** Nous considérons la théorie  $\mathbf{T}$  suivante, proposée dans la syntaxe CiME :

```
Eq(0,S(x)) -> False;
Eq(S(x),0) -> False;
Eq(S(x),S(y)) -> Eq(x,y);
Eq(x,x) -> True;
0+x -> x ;
S(x)+y -> S(x+y);
```

ainsi que la conjecture  $S(x)+0 = x$ . Une fois l'ordre déterminé<sup>16</sup>, CiME complète la théorie, le système résultat est :

```
- : (F,X) TRS = { x + 0 -> x,
                  True -> False,
                  S(x) -> x,
                  Eq(x,0) -> False,
                  Eq(0,x) -> False,
                  0 + x -> x,
                  Eq(x,x) -> False } (7 rules)
```

La complétion produit la règle  $True \rightarrow False$  qui contredit l'axiomatisation, cette règle est appelée témoin d'inconsistance.

<sup>15</sup>où  $\mathcal{F}$  est la conjecture à prouver

<sup>16</sup>Pour effectuer la complétion il est nécessaire de choisir un ordre dans le but d'orienter les équations produites lors de la complétion, le succès de l'algorithme dépend essentiellement de l'ordre choisit

## L'approche de G. Huet et J.M. Hullot

Cette seconde approche suppose que l'alphabet considéré peut être scindé en deux ensembles de symboles : les symboles constructeurs et les symboles définis.

**Définition 51** Soit  $\mathcal{F}$  un alphabet, un sous ensemble  $\mathcal{C}$  de  $\mathcal{F}$  est appelé ensemble de constructeurs pour une théorie  $\mathbf{T}$  si pour chaque terme  $t$  de  $\mathcal{T}(\mathcal{F})$ , il existe un terme  $u$  de  $\mathcal{T}(\mathcal{C})$  tel que  $t \equiv_{\mathbf{T}} u$ .

Nous dirons qu'un terme  $t$  est un terme constructeur si  $t \in \mathcal{T}(\mathcal{C})$ . Un ensemble de constructeurs est dit "libre" si et seulement si, pour tous termes distincts  $s$  et  $t$  de  $\mathcal{T}(\mathcal{C})$ ,  $s \not\equiv_{\mathbf{T}} t$ . G. Huet et J.M. Hullot considèrent des théories où tous les constructeurs sont libres, et représentées par un système de réécriture fini et convergent. L'axiomatisation est alors l'ensemble des formules suivantes :

$$Axiom = \{\forall \vec{x} \forall \vec{y} f(\vec{x}) = f(\vec{y}) \Rightarrow \vec{x} = \vec{y} \mid f \in \mathcal{C}\} \cup \{\forall \vec{x} \forall \vec{y} f(\vec{x}) \neq g(\vec{y}) \mid f, g \in \mathcal{C} f \neq g\}$$

$Axiom$  est une axiomatisation puisque  $\mathcal{T}(\mathcal{F}) / \equiv_{\mathbf{T}}$  est modèle de  $\mathbf{T} \cup Axiom$ . Par ailleurs, s'il existe un autre modèle  $\mathfrak{M}$  de  $\mathbf{T} \cup Axiom$  alors pour tout couple de termes  $(s, t)$ , par définition il existe un couple  $(s', t')$  de termes constructeurs tels que  $t \equiv_{\mathbf{T}} t'$ ,  $s' \equiv_{\mathbf{T}} s$ . Nécessairement  $\mathfrak{M}$ , puisque  $s$  et  $t$  sont identiques,  $s'$  et  $t'$  le sont aussi, nous avons donc montré que deux termes étaient identiques si et seulement si ils étaient égaux modulo  $\mathbf{T}$  donc  $\mathfrak{M}$  est isomorphe à  $\mathcal{T}(\mathcal{F}) / \equiv_{\mathbf{T}}$ .

En réalité, cette approche est équivalente à celle de D. Musser puisque l'axiomatisation utilisée permet de construire une fonction  $eq$  complètement définie et vérifiant les hypothèses de D. Musser.

## Réductibilité inductive

**Définition 52 (Réductibilité inductive)** Un terme est dit inductiblement réductible par rapport à un système de réécriture  $\mathcal{R}$  si toutes ses instances closes sont réductibles par  $\mathcal{R}$ .

Le problème de la réductibilité inductive est décidable pour un système de réécriture sans symbole AC ou si le système de réécriture est linéaire, mais cette propriété devient indécidable sinon [KNZ91b]. Notre lecteur pourra trouver dans [Jac96, CJ94, CJ97] une description de méthodes de décision de la réductibilité inductive. La dernière méthode que nous décrivons ne nécessite pas de prédicat  $eq$ , s'appuie sur une théorie où les constructeurs ne sont pas nécessairement libres, mais demande néanmoins que la théorie puisse être orientée en un système de réécriture fini et convergent sur les termes clos. L'axiomatisation utilise un prédicat appelé  $Red$ , tel que pour tout terme  $t$ ,  $Red(t)$  est vérifié si et seulement si  $t$  est inductiblement réductible.  $Axiom$  est alors définie par :

$$Axiom = \{s = t \Rightarrow Red(s) \text{ si } s \succ t \text{ Red}(t) \text{ sinon}\}$$

Montrons que  $Axiom$  est une axiomatisation :

1. Supposons un couple de termes  $(s, t)$  tel que pour toute substitution close  $\sigma$   $s\sigma \equiv_E t\sigma$ . Comme  $\mathbf{T}$  est un système convergent, nécessairement  $s\sigma$  et  $t\sigma$  ont la même forme normale. Soit  $s\sigma$  et  $t\sigma$  sont syntaxiquement égaux, soit  $s\sigma$  ou  $t\sigma$  est réductible et si  $s\sigma \succ t\sigma$  alors  $s\sigma$  est nécessairement réductible sinon les deux termes ne peuvent pas avoir la même forme normale.

2. Si par ailleurs il existe une autre structure de Herbrand  $\mathfrak{M}$  qui soit modèle de  $\mathbf{T} \cup \text{Axiom}$ , il s'agit de montrer que pour tout couple de terme tel que  $s \leftrightarrow^n t$  si et seulement si  $s$  et  $t$  sont dans la même classe d'équivalence pour  $\equiv_{\mathbf{T}}$ . La preuve se fait par récurrence sur la longueur de la dérivation  $\leftrightarrow^n$ .

- (a) si  $n = 0$  alors  $s$  et  $t$  sont syntaxiquement égaux et donc  $s \equiv_{\mathbf{T}} t$
- (b) supposons que si  $s \leftrightarrow^n t$  alors  $s \equiv_{\mathbf{T}} t$
- (c) si  $s \leftrightarrow^{n+1} t$  supposons que  $s \leftrightarrow^n s' \leftrightarrow t$  par hypothèse de récurrence  $s \equiv_{\mathbf{T}} s'$  et nécessairement  $s \equiv_{\mathbf{T}} t$ .

La déduction s'appuie encore sur une complétion de Knuth Bendix et l'inconsistance est donnée par le test de réductibilité inductive du système obtenu.

**Exemple 21** *Considérons la conjecture  $\mathcal{F} = x + x = 0$  dans la théorie  $\mathbf{T}$  suivante :*

$$0 + x \rightarrow x \tag{6.9}$$

$$S(x) + y \rightarrow S(x + y) \tag{6.10}$$

$$S(S(0)) \rightarrow 0 \tag{6.11}$$

*La complétion calcule un nouveau système qui contient les deux premières règles de la théorie ainsi que les trois suivantes*

$$x + S(x) \rightarrow S(0) \tag{6.12}$$

$$S(S(x)) \rightarrow x \tag{6.13}$$

$$x + x \rightarrow 0 \tag{6.14}$$

*Il s'agit maintenant de montrer que les nouvelles règles sont inductivement réductibles, nous n'en donnerons pas la preuve, elle peut être trouvée dans [Com94].*

Le principe de preuve par cohérence offre l'avantage d'être automatisable même si le processus de complétion diverge souvent, en contre-partie l'utilisateur n'a guère de contrôle sur ses preuves.

### 6.2.3 Récurrence par réécriture

Lorsque la théorie considérée est orientable en un système de réécriture qui termine, alors la relation de réécriture induit un ordre bien fondé  $\prec$  sur les termes qui peut alors être utilisé comme support de raisonnement pour la récurrence. Le principe général est de caractériser le modèle initial par des ensembles couvrants :

**Définition 53 (Ensemble couvrant)** *Un ensemble fini de termes  $\{t_i\}_{i \in I}$  irréductibles de sorte  $s$  est un ensemble couvrant pour  $s$  par rapport à  $\mathcal{R}$  et  $\prec$  si et seulement si pour tout terme clos  $g$  de sorte  $s$  il existe un terme  $t_i$  et une substitution  $\tau$  tels que  $g \equiv_{\mathbf{T}} t_i \tau$  et  $t_i \tau \prec g$ .*

**Définition 54 (Substitution couvrante)** *Soit  $\bigcup_{s_j \in S} \{t_i : s_j\}_{i \in I} t_i$  une union d'ensembles couvrants pour l'ensemble des sortes  $s_j$ , une substitution couvrante  $\sigma$  pour un terme donné  $t$  est une substitution qui remplace chaque variable de  $t$  de sorte  $s_j$  par un élément de  $\{t_i : s_j\}_{i \in I}$ . Un ensemble couvrant de substitutions pour un terme  $t$  est simplement l'ensemble de toutes les substitutions couvrantes possibles pour  $t$ .*

Les ensembles couvrants vont permettre de prouver des théorèmes par récurrence dans l'ensemble des termes irréductibles plutôt que dans  $\mathcal{T}(\mathcal{F})$  tout entier. Nous présentons ici l'approche de U.S. Reddy présentée dans [Red90]. Considérons  $\mathcal{R}$  un système de réécriture terminant, une équation  $e = e'$  ainsi que  $\prec$  un ordre bien fondé sur les termes compatibles avec l'ordre  $\prec_{\mathbf{T}}$  induit par la relation de réécriture.  $\mathcal{T}(\mathcal{F}) / \equiv_{\mathbf{T}}$  est modèle de  $e = e'$  si et seulement si

1.  $\{\sigma_i\}_I$  est un ensemble couvrant de substitutions
2.  $\forall \sigma \in \{\sigma_i\}_I (\forall \theta \{e\theta, e'\theta\} \prec_{\mathbf{T}}^{17} \{e\sigma_i, e'\sigma_i\} e\theta = e'\theta) \Rightarrow e\sigma_i = e'\sigma_i$

La seconde condition va imposer que les hypothèses appliquées soient plus petites que la conjecture, c'est ce qui traduit la récurrence.

**Exemple 22** Reprenons la théorie définissant + sur les entiers :

$$0 + x = x \quad (6.15)$$

$$\text{Succ}(x) + y = \text{Succ}(x + y) \quad (6.16)$$

ainsi que la conjecture  $0 + x = x$ . L'ensemble  $\{0, \text{Succ}(y)\}$  est un ensemble couvrant pour  $x$  qui induit l'ensemble couvrant de substitution  $\{[x \leftarrow 0], [x \leftarrow \text{Succ}(y)]\}$  Il s'agit de montrer que :  $\forall \sigma \in \{[x \leftarrow 0], [x \leftarrow \text{Succ}(y)]\} (\forall \theta \{(0+x)\theta, x'\theta\} \prec_{\mathbf{T}} \{(0+x)\sigma_i, x\sigma_i\} (0+x)\theta = x\theta) \Rightarrow (0+x)\sigma_i = x\sigma_i$

Soit deux buts à prouver :

1.  $(\forall \theta \{(0+x)\theta, x\theta\} \prec_{\mathbf{T}} \{0+0, 0\} (0+x)\theta = x\theta) \Rightarrow (0+0 = 0)$  Comme  $(0+0) \rightarrow_{6.15} 0$ , le premier but est prouvé
2. il reste  $(\forall \theta \{(0+x)\theta, x\theta\} \prec_{\mathbf{T}} \{0+\text{Succ}(y), \text{Succ}(y)\} (0+x)\theta = x\theta) \Rightarrow (0+\text{Succ}(y) = \text{Succ}(y))$   
Or les substitutions  $\theta$  vérifiant  $\{(0+x)\theta, x\theta\} \prec_{\mathbf{T}} \{0+\text{Succ}(y), \text{Succ}(y)\}$  sont  $[x \leftarrow 0]$  et  $[x \leftarrow y]$  ce qui nous amène à montrer que les deux égalités  $(0+0 = 0)$  et  $(0+y = y)$  (qui sont donc les hypothèses de récurrence) implique  $(0+\text{Succ}(y) = \text{Succ}(y))$ .

$$(0+\text{Succ}(y) = \text{Succ}(y)) \rightarrow_{6.16} (\text{Succ}(0+y) = \text{Succ}(y)) \rightarrow_{(0+y=y)} (\text{Succ}(y) = \text{Succ}(y))$$

Le second but est montré.

L'avantage de cette méthode est l'automatisation du raisonnement par récurrence par rapport aux méthodes de récurrence explicite, par ailleurs il n'est pas nécessaire de chercher une axiomatisation du plus petit modèle de la théorie comme dans les preuves par cohérence. En revanche, il n'existe pas de méthodes pour trouver automatiquement les ensembles couvrants et le processus de preuve diverge souvent. C'est pourquoi A. bouhoula, E. Kounalis et M. Rusinowitch [BKR95] ainsi que D. Kapur, P. Narendran et H. Zhang ont proposé dans et [KNZ91a] d'utiliser des ensembles tests qui sont un raffinement des ensembles couvrants. La méthode de récurrence par réécriture avec ensemble tests a ensuite été reprise par A. Bouhoula [BR95a, Bou97] dans les systèmes conditionnels, c'est la base du prouveur *Spike* [BR95b].

<sup>17</sup>où  $\prec_{\mathbf{T}}$  est l'extension multi-ensemble de  $\prec$



7

## Preuves de propriétés initiales

## Sommaire

<b>7.1</b>	<b>Principe de l'approche</b> . . . . .	<b>86</b>
7.1.1	Exemple introductif . . . . .	86
7.1.2	Système de preuve . . . . .	88
7.1.3	Correction de l'algorithme . . . . .	89
7.1.4	Un exemple détaillé . . . . .	91
<b>7.2</b>	<b>Preuves de propriétés initiales par complétion d'automate</b> . . .	<b>93</b>
7.2.1	Modélisation du système . . . . .	93
7.2.2	Principe général . . . . .	95
7.2.3	Reprenons notre exemple . . . . .	95
7.2.4	Limite de l'approche . . . . .	96
<b>7.3</b>	<b>Vers l'interprétation abstraite</b> . . . . .	<b>96</b>
<b>7.4</b>	<b>Et la complétude réfutationnelle ?</b> . . . . .	<b>97</b>

## 7.1 Principe de l'approche

Le problème qui nous intéresse maintenant est de trouver une méthode, la plus automatique possible pour prouver des propriétés négatives de la forme  $\forall x_1 \dots x_n \neg(t_1 = t_2)$  où  $x_1, \dots, x_n$  sont les variables qui apparaissent dans  $t_1, t_2$ , sur le plus petit modèle de Herbrand d'une théorie équationnelle  $\mathbf{T}$  donnée. La première partie de ce chapitre propose un système de preuve de propriétés initiales, la seconde partie montre comment nous pouvons utiliser la complétion d'automates pour automatiser cet algorithme.

**Définition 55 (Propriétés initiales)** *Soit  $\mathcal{F}$  un alphabet,  $\mathbf{T}$  une théorie,  $t_1, t_2$  deux termes,  $\mathcal{D}$  une diséquation c'est à dire une formule de la forme  $\mathcal{D} = \forall x_1 \dots x_n \neg(t_1 \simeq t_2)$ <sup>18</sup> est une propriété initiale si c'est une formule vraie dans  $\mathcal{T}(\mathcal{F}) / \equiv_{\mathbf{T}}$ , c'est à dire si toutes ses instances closes le sont.*

Notre problème peut se voir comme un cas particulier de disunification [Com91], la disunification s'attache à trouver des solutions aux diséquations, c'est un problème qui est assez peu étudié. S. Limet et P. Réty utilisent dans [LR98] des grammaires synchronisées de T-uples d'arbres ou TTSG [GRS01, LR97] pour énumérer toutes les solutions d'une diséquation  $t_1 \neq t_2$ , cet algorithme est un procédure de semi-décision pour prouver l'existence de solutions. Nous proposons une méthode qui, quand elle termine, prouve qu'une diséquation est vraie dans le plus petit modèle de Herbrand de la théorie. Nous avons déjà proposé un premier algorithme dans [GVTT02] que nous améliorons ici, avant de proposer une approche par automate.

### 7.1.1 Exemple introductif

Nous cherchons à mettre en place un système de preuve par réécriture des propriétés initiales basé sur le principe de descente infinie ( dû à Fermat ). Il s'agit d'un raisonnement par l'absurde : Etant donné un problème portant sur un ensemble sur lequel il existe un ordre bien fondé, on suppose d'abord que ce problème a une solution ; puis on démontre qu'il existe alors une autre solution strictement plus petite, et la contradiction vient du fait qu'il ne peut exister de suite

<sup>18</sup>par commodité, nous noterons  $\mathcal{D} = t_1 \neq t_2$

infinie strictement décroissante. Nous allons nous inspirer de ce raisonnement : plaçons nous dans  $\mathbb{N}$  et supposons que nous voulions montrer qu'une propriété  $P$  est fausse pour tout entier, intuitivement, notre raisonnement va consister à montrer que  $P$  est fausse pour 0 et que si il existait un entier  $n > 0$  vérifiant  $P$  alors  $P$  serait vraie pour au moins un entier inférieur à  $n$ . Comme  $P$  est fausse pour 0,  $P$  est fausse pour tous les entiers.

Pour commencer, nous proposons un exemple simple basé sur la définition de deux fonctions : *even* et *odd*.

Considérons la conjecture  $\forall x \text{even}(x) \neq \text{odd}(x)$  dans la théorie suivante :

$$\text{even}(0) = \text{True} \quad (7.1)$$

$$\text{odd}(0) = \text{False} \quad (7.2)$$

$$\text{even}(\text{Succ}(x)) = \text{odd}(x) \quad (7.3)$$

$$\text{odd}(\text{Succ}(x)) = \text{even}(x) \quad (7.4)$$

$$(7.5)$$

Cette théorie forme une spécification suffisamment complète. Intuitivement, une spécification suffisamment complète est un ensemble de règles qui décrit complètement un ensemble de fonctions.

**Définition 56 (Suffisante complétude)** *Considérons une théorie  $\mathbf{T}$  sur un ensemble des symboles partagé en symboles constructeurs et définis.  $\mathbf{T}$  est une spécification suffisamment complète si tout terme clos est équivalent suivant  $\equiv_{\mathbf{T}}$  à un terme clos construit uniquement avec des symboles constructeurs.*

Le problème de la suffisante complétude est indécidable en général, mais il existe des classes de définitions pour lesquelles le problème admet des solutions. En particulier, ce problème est décidable pour un système linéaire  $\mathcal{R}$  terminant et convergent et tel que  $\mathcal{R}$  préserve les constructeurs : pour toute règle  $l \rightarrow r$  telle que  $l \in \mathcal{T}(\mathcal{C}, \mathcal{X})$ , on a  $r \in \mathcal{T}(\mathcal{C}, \mathcal{X})$ . A. Bouhoula a proposé une méthode pour vérifier la suffisante complétude dans les spécifications paramétrées [Bou96].

Pour prouver la conjecture, il suffit donc de s'intéresser aux cas  $x = 0$ ,  $x = \text{Succ}(y)$ . le premier but à prouver est

$$\text{even}(0) \neq \text{odd}(0)$$

ce qui se réduit en  $\text{True} \neq \text{False}$  en utilisant 7.1 et 7.2. Le premier but est prouvé.

Il reste

$$(\text{even}(\text{Succ}(y)) = \text{odd}(\text{Succ}(y))) \Rightarrow (\text{even}(y) = \text{odd}(y))$$

par application des règles 7.3 et 7.3 cette implication se réduit en  $(\text{odd}(y) = \text{even}(y)) \Rightarrow (\text{even}(y) = \text{odd}(y))$  qui est une trivialité.

Cet exemple se conclut uniquement en examinant tous les cas possibles pour  $x$  en utilisant une substitution couvrante pour  $x$  car il n'y avait pas de variable de récurrence. Nous donnons maintenant une définition qui va permettre de savoir sélectionner les schémas de récurrence.

**Définition 57 (Positions de récurrence [Bou97])** *Soit  $\mathcal{R}$  un système de réécriture,  $\mathcal{C}$  une (dis)-équation, l'ensemble des variables de récurrence de  $\mathcal{C}$  est le plus petit ensemble qui contient*

- toutes les variables de sortes finitaires (voir définition 10)
- et toutes les variables  $x$  apparaissant à la position  $u$  d'un sous terme  $t$  de  $\mathcal{C}$ , et telles que  $t$  est unifiable avec une règle  $l \rightarrow r$  de  $\mathcal{R}$  et vérifiant l'une des conditions suivantes :

- le sous terme à la position  $u$  de  $l$  n'est pas une variable
- le sous terme à la position  $u$  de  $l$  est une variable  $y$  et  $l$  n'est pas linéaire en  $y$ .

Nous noterons  $t[x]$ , le terme  $t$  s'il contient une variable de récurrence  $x$ .

**Exemple 23** *Considérons par exemple l'équation  $Plus(Plus(x, y), 0) = Plus(x, y)$  dans la théorie définissant le symbole  $Plus$  :*

$$Plus(0, y) \rightarrow x \quad (7.6)$$

$$Plus(Succ(x), y) \rightarrow Succ(Plus(x, y)) \quad (7.7)$$

La seule variable de récurrence est  $x$ .

### 7.1.2 Système de preuve

Notre première approche est un système d'inférence basé sur ce raisonnement que nous avons présenté dans [GVTT02].

Nous supposons que

1. Tous les constructeurs sont libres
2.  $\mathcal{R}$  forme une spécification suffisamment complète
3. Chaque (dis)-équation à prouver possède au plus une variable de récurrence

Nous présentons notre système de procédure de preuve en figure 7.1.2 sous la forme d'un système d'inférence avec la structure de données  $(\mathcal{E}, Hr, C_-, C_+)$  où

- $\mathcal{E}$  est un ensemble de règles  $l \rightarrow r$  telles que  $l = r$  est soit une règle de  $\mathcal{R}$ , soit un théorème inductif
- $Hr$  est un ensemble d'hypothèses de récurrence
- $C_+$  un ensemble de conjectures positives
- $C_-$  un ensemble de conjectures négatives

Avant toute chose, nous renommons les variables de  $C_-$  de sorte que toutes les conjectures aient des variables distinctes des autres conjectures. Cet algorithme utilise une fonction appelée *cover* dont le but est d'instancier de toutes les manières possibles (à l'aide d'ensemble couvrant) les termes n'ayant plus de variables de récurrence.

**Définition 58 (Skolémisation)** *La skolémisation consiste à éliminer des quantificateurs existentiels en remplaçant les variables existentielles par des termes constitués de constructeurs appliqués à de nouvelles variables.*

**Définition 59 (Cover)** *Soit  $t \in \mathcal{T}(\mathcal{F}, \mathcal{X})$ ,  $cover(t)$  est l'ensemble de tous les termes obtenus par application d'une substitution couvrante sur chacune des variables de  $t$ . Dans  $cover(t)$ , toutes les variables sont skolémisées, une variable  $x$  skolémisée sera notée  $\bar{x}$  et sera donc considérée comme une constante.*

**Exemple 24** *Considérons la spécification sur l'alphabet  $\{Plus : nat \rightarrow nat, S : nat \rightarrow nat, 0 : nat\}$  :*

$$Plus(0, y) \rightarrow y \quad (7.8)$$

$$Plus(S(x), y) \rightarrow S(Plus(x, y)) \quad (7.9)$$

Le terme  $Plus(0, y)$  n'a pas de variable de récurrence. La variable  $y$  est de sorte  $nat$ , nous prenons  $\{0, S(\bar{y})\}$  comme ensemble couvrant pour  $nat$ .

$$cover[Plus(0, y)] = \{Plus(0, 0), Plus(0, S(\bar{y}))\}$$

Enfin pour appliquer notre raisonnement (descente infinie) nous avons besoin de séparer pour toute variable de récurrence  $x$  de sorte  $s$  et tout ensemble couvrant  $C$  pour  $s$ , afin de distinguer les cas de bases  $C_b$  et les cas de récurrence  $C_{rec}$ .

- $C_b = C$  et  $C_{rec} = \emptyset$  si  $s$  est de sorte finitaire
- sinon  $C_b$  contient tous les termes clos de  $C$  et  $C_{rec} = C \setminus C_b$

Par suite, si  $x$  est une variable de récurrence de sorte  $s$ ,  $C$  un ensemble couvrant pour  $s$  et  $t$  un terme, l'ensemble  $t[x\sigma_b]$  (respectivement  $t[x\sigma_{rec}]$ ) désignera l'ensemble des termes obtenu en substituant les occurrences de  $x$  dans  $t$  par un élément de  $C_b$  (respectivement de  $C_{rec}$ ).

**Définition 60** Une  $\mathcal{I}$ -dérivation est une séquence  $(\mathcal{E}_0, Hr_0, C_{0,-}, C_{0,+}) \vdash_{\mathcal{I}} (\mathcal{E}_1, Hr_1, C_{1,-}, C_{1,+}) \vdash_{\mathcal{I}} \dots \vdash_{\mathcal{I}} (\mathcal{E}_n, Hr_n, C_{n,-}, C_{n,+})$ . Nous dirons qu'une  $\mathcal{I}$ -dérivation est réussie si elle se termine par  $(\mathcal{E}_n, Hr_n, \emptyset, \emptyset)$ .

### 7.1.3 Correction de l'algorithme

#### THEOREME 6: Correction de $\mathcal{I}$

Soit  $\mathcal{E}$  une théorie équationnelle orientable en un système de réécriture contenant

1. une spécification suffisamment complète des symboles définis
2. un ensemble de théorèmes inductifs

Une diséquation  $t_1 \neq t_2$  est une propriété initiale s'il existe une  $\mathcal{I}$ -dérivation réussie à partir de  $(\mathcal{E}, \emptyset, \{t_1 \neq t_2\}, \emptyset)$

#### Démonstration: $\triangleright$

Nous allons montrer par récurrence sur  $k$  que si  $(\mathcal{E}, Hr, C_-, C_+) \vdash_{\mathcal{I}}^k (\mathcal{E}, Hr, \emptyset, \emptyset)$  alors  $C_-$  est vraie dans  $\mathcal{T}(\mathcal{F}, \mathcal{X})/\equiv_{\mathcal{E}}$  et que  $C_+$  est vraie dans  $\mathcal{T}(\mathcal{F}, \mathcal{X})/\equiv_{\mathcal{E} \cup Hr}$

1. - Soit  $(\mathcal{E}, Hr, C_-, \emptyset) \vdash_{\mathcal{I}} (\mathcal{E}, Hr, \emptyset, \emptyset)$  et nécessairement  $C_-$  est de la forme  $\{f(\vec{t}_1) \neq g(\vec{t}_2)\}$  avec  $f, g \in \mathcal{C}$ ,  $f \neq g$  et la seule règle qui s'applique est **elim -**, alors par définition d'un terme constructeur  $f(\vec{t}_1)$  et  $g(\vec{t}_2)$  sont dans deux classes d'équivalences distinctes de  $\mathcal{T}(\mathcal{F}, \mathcal{X})/\equiv_{\mathcal{E}}$ .
- Soit  $(\mathcal{E}, Hr, \emptyset, C_+) \vdash_{\mathcal{I}} (\mathcal{E}, Hr, \emptyset, \emptyset)$  et soit c'est la règle **elim +** qui s'applique et en particulier  $C_+$  est vraie dans n'importe quel modèle de  $\mathcal{E}$ , soit c'est la règle **simplify2** qui s'applique,  $C_+$  est de la forme  $\{t_1 = t_2\}$  et  $Hr$  contient  $\{t_1 = t_2\}$  donc  $C_+$  est vraie sous les hypothèses  $Hr$ .
2. Nous supposons que pour toute conjecture  $P$ , si  $(\mathcal{E}, Hr, P, \emptyset) \vdash_{\mathcal{I}}^n (\mathcal{E}, Hr^n, \emptyset, \emptyset)$  alors  $P$  est une propriété de  $\mathcal{T}(\mathcal{F}, \mathcal{X})/\equiv_{\mathcal{E}}$  sous les hypothèses  $Hr$ . Nous raisonnons par cas sur la première règle utilisée lors de la dérivation :

**simplify1**  $(\mathcal{E}, \emptyset, C_- \cup \{t_1 \neq t_2\}, C_+) \vdash_{\mathcal{I}} (\mathcal{E}, \emptyset, C_- \cup \{t'_1 \neq t_2\},) \vdash_{\mathcal{I}}^n (\mathcal{E}, Hr, \emptyset, \emptyset)$  avec  $t_1 \rightarrow_{\mathcal{E}} t'_1$ .

Si  $t_1 \rightarrow_{\mathcal{E}} t'_1$  alors  $t_1$  et  $t'_1$  sont dans la même classe d'équivalence dans  $\mathcal{T}(\mathcal{F}, \mathcal{X})/\equiv_{\mathcal{E}}$  et nécessairement  $t_1 \neq t_2$  si et seulement si  $t'_1 \neq t_2$ , par hypothèse de récurrence  $C_- \cup \{t'_1 \neq t_2\}$  est une propriété de  $\mathcal{T}(\mathcal{F}, \mathcal{X})/\equiv_{\mathcal{E}}$  donc  $C_- \cup \{t_1 \neq t_2\}$  est aussi une propriété de  $\mathcal{T}(\mathcal{F}, \mathcal{X})/\equiv_{\mathcal{E}}$ .

1. des règles de simplification par réécriture

**simplify1**

$$(\mathcal{E}, Hr, C_- \cup \{t_1 \neq t_2\}, C_+) \vdash_{\mathcal{I}} (\mathcal{E}, Hr, C_- \cup \{t'_1 \neq t_2\}, C_+) \text{ si } t_1 \rightarrow_{\mathcal{E}} t'_1$$

**simplify2**

$$(\mathcal{E}, Hr \cup \{t_1 = t_2\}, C_-, C_+ \cup \{t_1 = t_2\}) \vdash_{\mathcal{I}} (\mathcal{E}, Hr \cup \{t_1 = t_2\}, C_-, C_+)$$

**simplify3**

$$(\mathcal{E}, Hr \cup \{t_1 = t_2\}, C_-, C_+) \vdash_{\mathcal{I}} (\mathcal{E}, Hr \cup \{t'_1 = t_2\}, C_-, C_+) \text{ si } t_1 \rightarrow_{\mathcal{E}} t'_1$$

2. une règle d'élimination des conjectures négatives :

**elim -**

$$(\mathcal{E}, Hr, C_- \cup \{f(\vec{t}_1) \neq g(\vec{t}_2)\}, C_+) \vdash_{\mathcal{I}} (\mathcal{E}, Hr, C_-, C_+) \text{ (si } f, g \in \mathcal{C}, f \neq g)$$

3. une règle d'élimination des conjectures positives (tautologies) :

**elim +**

$$(\mathcal{E}, Hr, C_-, C_+ \cup \{t = t\}) \vdash_{\mathcal{I}} (\mathcal{E}, Hr, C_-, C_+)$$

4. une règle de simplification des conjectures positives :

**simp +**

$$(\mathcal{E}, Hr, C_-, C_+ \cup \{f(\vec{t}_1) = f(\vec{t}_2)\}) \vdash_{\mathcal{I}} (\mathcal{E}, Hr, C_-, C_+ \cup \{\vec{t}_1 = \vec{t}_2\}) \text{ (si } f \in \mathcal{C})$$

5. une règle de décomposition sur les constructeurs :

**décomp** ( $f \in \mathcal{C}$ )

$$(\mathcal{E}, Hr \cup \{f(\vec{t}_1) = f(\vec{t}_2)\}, C_-, C_+) \vdash_{\mathcal{I}} (\mathcal{E}, Hr \cup \{f(\vec{t}_1) = f(\vec{t}_2)\} \cup \{\vec{t}_1 = \vec{t}_2\}, C_-, C_+)$$

6. la règle d'application du raisonnement par récurrence, si la conjecture contient une variable de récurrence  $x$  :

**rec**

$$(\mathcal{E}, Hr, C_- \cup \{t_1[x] = t_2[x]\}, C_+) \vdash_{\mathcal{I}} (\mathcal{E}, Hr \cup Hr', C_- \cup C'_-, C_+ \cup C'_+)$$

où

$$C'_- = \text{cover}[t_1[x\sigma_b] \neq t_2[x\sigma_b]],$$

$$Hr' = \text{cover}[t_1[x\sigma_{rec}] = t_2[x\sigma_{rec}]]$$

$$C'_+ = \text{cover}[t_1[x \mapsto \bar{x}] = t_2[x \mapsto \bar{x}]]$$

FIG. 7.1 – Système d'inférence  $\mathcal{I}$

**elim** -  $(\mathcal{E}, \emptyset, C_- \cup \{f(\vec{t}_1) \neq g(\vec{t}_2)\}, C_+) \vdash_{\mathcal{T}} (\mathcal{E}, \emptyset, C_-, C_+) \vdash_{\mathcal{T}}^n (\mathcal{E}, Hr, \emptyset, \emptyset)$  avec  $f, g \in \mathcal{C}$ ,  $f \neq g$ . Nous avons déjà vu que  $f(\vec{t}_1)$  et  $g(\vec{t}_2)$  sont dans deux classes d'équivalence distinctes de  $\mathcal{T}(\mathcal{F}, \mathcal{X})/\equiv_{\varepsilon}$  si  $f, g \in \mathcal{C}$ ,  $f \neq g$  donc  $f(\vec{t}_1) \neq g(\vec{t}_2)$  est bien une propriété de  $\mathcal{T}(\mathcal{F}, \mathcal{X})/\equiv_{\varepsilon}$ , par hypothèse de récurrence  $C_-$  est bien une propriété de  $\mathcal{T}(\mathcal{F}, \mathcal{X})/\equiv_{\varepsilon}$ . Finalement  $C_- \cup \{f(\vec{t}_1) \neq g(\vec{t}_2)\}$  est une propriété de  $\mathcal{T}(\mathcal{F}, \mathcal{X})/\equiv_{\varepsilon}$ .

**simp** +

$$(\mathcal{E}, Hr, C_-, C_+ \cup \{f(\vec{t}_1) = f(\vec{t}_2)\}) \vdash_{\mathcal{T}} (\mathcal{E}, Hr, C_-, C_+ \cup \{\vec{t}_1 = \vec{t}_2\}) \text{ (si } f \in \mathcal{C})$$

**elim** +  $(\mathcal{E}, \emptyset, C_-, C_+ \cup \{t = t\}) \vdash_{\mathcal{T}} (\mathcal{E}, \emptyset, C_-, C_+) \vdash_{\mathcal{T}}^n (\mathcal{E}, Hr, \emptyset, \emptyset)$ . Par hypothèse de récurrence  $C_-$  est une propriété de  $\mathcal{T}(\mathcal{F}, \mathcal{X})/\equiv_{\varepsilon}$  et  $C_+$  est vrai dans  $\mathcal{T}(\mathcal{F}, \mathcal{X})/\equiv_{\varepsilon \cup Hr}$ , clairement  $C_+ \cup \{t = t\}$  est encore vraie dans  $\mathcal{T}(\mathcal{F}, \mathcal{X})/\equiv_{\varepsilon \cup Hr}$ .

**rec**  $(\mathcal{E}, \emptyset, C_- \cup \{t_1[x] = t_2[x]\}, C_+) \vdash_{\mathcal{T}} (\mathcal{E}, Hr, C_- \cup C'_-, C_+ \cup C'_+) \vdash_{\mathcal{T}}^n (\mathcal{E}, Hr^n, \emptyset, \emptyset)$

$$C'_- = \text{cover}[t_1[x\sigma_b] \neq t_2[x\sigma_b]],$$

$$Hr' = \text{cover}[t_1[x\sigma_{rec}] = t_2[x\sigma_{rec}]]$$

$$C'_+ = \text{cover}[t_1[x \mapsto \bar{x}] = t_2[x \mapsto \bar{x}]]$$

Nous avons  $(\mathcal{E}, Hr, C_- \cup C'_-, C_+) \vdash_{\mathcal{T}}^n (\mathcal{E}, Hr, \emptyset, \emptyset)$ . remarquons que nous pouvons séparer cette dérivation en deux puisque si la règle **rec** est appliquée, comme on ne considère que des conjectures avec une seule variable de récurrence, **rec** ne peut pas être appliquée de nouveau et donc les ensembles  $C_+$  et  $C_-$  n'interagissent plus, il suffit donc de montrer que nous pouvons "vider" les deux ensembles de conjectures  $C_-$  et  $C_+$ .

(a)  $(\mathcal{E}, \emptyset, C_- \cup C'_-, \emptyset) \vdash_{\mathcal{T}}^n (\mathcal{E}, Hr, \emptyset, \emptyset)$  car  $Hr$  n'entre pas dans la dérivation de conjectures négatives, par hypothèse de récurrence  $C_- \cup C'_-$  est une propriété de  $\mathcal{T}(\mathcal{F}, \mathcal{X})/\equiv_{\varepsilon}$

(b)  $(\mathcal{E}, Hr, \emptyset, C_+ \cup C'_+) \vdash_{\mathcal{T}}^n (\mathcal{E}, Hr, \emptyset, \emptyset)$  les seules règles qui s'appliquent sur  $C_+ \cup C'_+$  sont **simplify2**, et **elim+** puisqu'il n'y a plus de variables libres donc chaque équation se réécrit en une tautologie donc  $C_+ \cup C'_+$  sont vraies sous les hypothèses de récurrence  $Hr$ , ce qui revient à l'étape d'induction si  $P(x+1)$  est vraie alors nous avons déjà  $P(x)$ , pour conclure il faut encore que  $\text{cover}[t_1[x\sigma_b] \neq t_2[x\sigma_b]]$  soit une propriété de  $\mathcal{T}(\mathcal{F}, \mathcal{X})/\equiv_{\varepsilon}$ , ce qui vient d'être montré.

◁

#### 7.1.4 Un exemple détaillé

Nous considérons la théorie **T** suivante

$$\text{Plus}(0, x) = x \tag{7.10}$$

$$\text{Plus}(\text{Succ}(x), y) = \text{Succ}(\text{Plus}(x, y)) \tag{7.11}$$

$$\text{Moins}(0, x) = 0 \tag{7.12}$$

$$\text{Moins}(x, 0) = x \tag{7.13}$$

$$\text{Moins}(\text{Succ}(x), \text{Succ}(y)) = \text{Moins}(x, y) \tag{7.14}$$

Nous allons montrer que dans  $\mathcal{T}(\mathcal{F}, \mathcal{X})/\equiv_{\varepsilon}$

$$\forall x \forall y \text{Plus}(x, \text{S}(y)) \neq \text{Moins}(x, y) \tag{7.15}$$

Remarquons tout d'abord que cette affirmation n'est pas vraie dans tous les modèles  $\mathcal{E}$  : considérons le modèle  $\mathcal{H}$  défini sur le domaine réduit à  $\{0, S(0)\}$  et vérifiant :  $S(S(0)) = 0$ . Prenons  $x = 0$  et  $y = S(0)$ , (7.22) devient  $Plus(0, S(S(0))) \neq Moins(0, S(0))$  qui se réécrit en  $S(S(0)) \neq 0$  ce qui est faux dans  $\mathcal{H}$ .

Nous allons montrer que (7.22) est une propriété de  $\mathcal{T}(\mathcal{F}, \mathcal{X})/\equiv_{\mathcal{E}}$  en utilisant notre système d'inférence. Pour faire cette preuve nous utiliserons la commutativité de *Plus* ainsi que le lemme suivant :

$$Moins(x, S(y)) = Moins(Moins(x, y), S(0)) \quad (7.16)$$

qui est un théorème inductif.

Nous partons de

- $\mathcal{E} = \mathbf{T} \cup \{Moins(u, S(v)) \leftrightarrow Moins(Moins(u, v), S(0)), Plus(w, t) \leftrightarrow Plus(t, w)\}$
- $Hr = \emptyset$
- $C_- = \{Plus(x, S(y)) \neq Moins(x, y)\}$
- $C_+ = \emptyset$

D'après la définition 57,  $x$  est la seule variable de récurrence pour  $Plus(x, S(y)) \neq Moins(x, y)$ .  $x$  est de sorte *nat*,  $C_b = \{0\}$  and  $C_{rec} = \{S(\bar{x})\}$ .

La seule règle qui s'applique est celle du raisonnement par récurrence **rec** sur la variable  $x$  :

- $Hr = \{Plus(S(\bar{x}), S(0)) = Moins(S(\bar{x}), 0), Plus(S(\bar{x}), S(S(\bar{y}))) = Moins(S(\bar{x}), S(\bar{y}))\}$
- $C_- = \{Plus(0, S(0)) \neq Moins(0, 0), Plus(0, S(S(\bar{y}))) \neq Moins(0, S(\bar{y}))\}$
- $C_+ = \{Plus(\bar{x}, S(0)) = Moins(\bar{x}, 0), Plus(\bar{x}, S(S(\bar{y}))) = Moins(\bar{x}, S(\bar{y})), \}$

La règle **simplify1** peut être utilisée sur  $C_-$  qui devient alors  $\{S(0) \neq 0, S(S(\bar{y})) \neq 0\}$  ce qui se réduit en  $\emptyset$  grâce à la règle **elim-**.

- $Hr = \{Plus(S(\bar{x}), S(0)) = Moins(S(\bar{x}), 0), Plus(S(\bar{x}), S(S(\bar{y}))) = Moins(S(\bar{x}), S(\bar{y}))\}$
- $C'_- = \emptyset$
- $C_+ = \{Plus(\bar{x}, S(0)) = Moins(\bar{x}, 0), Plus(\bar{x}, S(S(\bar{y}))) = Moins(\bar{x}, S(\bar{y})), \}$

Maintenant nous utilisons **simplify3** pour réduire  $Hr$  :

$$Plus(S(\bar{x}), S(0)) = Moins(S(\bar{x}), 0) \rightarrow_{\mathbf{T}} S(Plus(\bar{x}), S(0)) = Moins(S(\bar{x}), 0)$$

$$S(Plus(\bar{x}), S(0)) = Moins(S(\bar{x}), 0) \rightarrow_{\mathcal{E}} S(Plus(S(0), \bar{x})) = Moins(S(\bar{x}), 0)$$

$$S(Plus(S(0), \bar{x})) = Moins(S(\bar{x}), 0) \rightarrow_{\mathbf{T}} S(S(Plus(0, \bar{x}))) = Moins(S(\bar{x}), 0)$$

$$S(S(Plus(0, \bar{x}))) = Moins(S(\bar{x}), 0) \rightarrow_{\mathbf{T}} S(S(\bar{x})) = Moins(S(\bar{x}), 0)$$

$$\text{et donc } S(S(\bar{x})) = Moins(S(\bar{x}), 0) \rightarrow_{\mathbf{T}} S(S(\bar{x})) = S(\bar{x})$$

Ensuite, nous utilisons **simplify2** pour réduire  $C_+$  :

$$(Plus(\bar{x}, S(0)) = Moins(\bar{x}, 0)) \rightarrow_{\mathcal{R}, \mathcal{E}} (S(\bar{x}) = \bar{x})$$

- $Hr = \{Plus(S(\bar{x}), S(0)) = Moins(S(\bar{x}), 0), Plus(S(\bar{x}), S(S(\bar{y}))) = Moins(S(\bar{x}), S(\bar{y}))\}$
- $C'_- = \emptyset$
- $C_+ = \{S(\bar{x}) = \bar{x}, Plus(\bar{x}, S(S(\bar{y}))) = Moins(\bar{x}, S(\bar{y}))\}$

L'application de **constr** donne

- $Hr = \{S(\bar{x}) = \bar{x}, Plus(S(\bar{x}), S(S(\bar{y}))) = Moins(S(\bar{x}), S(\bar{y}))\}$
- $C'_- = \emptyset$
- $C_+ = \{S(\bar{x}) = \bar{x}, Plus(\bar{x}, S(S(\bar{y}))) = Moins(\bar{x}, S(\bar{y}))\}$

La règle **simplify2** nous permet de conclure la première partie de  $C_+$ .

- $Hr = \{S(\bar{x}) = \bar{x}, Plus(S(\bar{x}), S(S(\bar{y}))) = Moins(S(\bar{x}), S(\bar{y}))\}$
- $C'_- = \emptyset$
- $C_+ = \{Plus(\bar{x}, S(S(\bar{y}))) = Moins(\bar{x}, S(\bar{y}))\}$

Pour finir la preuve, nous utilisons la séquence de réécriture suivante, qui correspond à l'application successive de la règle **simplify2**

$$\begin{aligned}
 Moins(\bar{x}, S(\bar{y})) & \\
 & \rightarrow_{\mathcal{E}} Moins(Moins(\bar{x}, \bar{y}), S(0)) \\
 & \rightarrow_{Hr} Moins(Plus(S(\bar{x}), S(S(\bar{y}))), S(0)) \\
 & \rightarrow_{\mathcal{E}} Moins(S(Plus(\bar{x}, S(S(\bar{y})))), S(0)) \\
 & \rightarrow_{\mathcal{E}} Moins(Plus(\bar{x}, S(S(\bar{y}))), 0) \\
 & \rightarrow_{\mathcal{E}} Plus(\bar{x}, S(S(\bar{y})))
 \end{aligned}$$

Nous terminons par

- $Hr = \{S(\bar{x}) = \bar{x}, Plus(S(\bar{x}), S(S(\bar{y}))) = Moins(S(\bar{x}), S(\bar{y}))\}$
- $C'_- = \emptyset$
- $C_+ = \{Plus(\bar{x}, S(S(\bar{y}))) = Plus(\bar{x}, S(S(\bar{y})))\}$

enfin l'application de la règle **elim+** permet de conclure la preuve, nous puisque aboutissons

à

- $Hr = \{S(\bar{x}) = S(\bar{x}), Plus(S(\bar{x}), S(S(\bar{y}))) = Moins(S(\bar{x}), S(\bar{y}))\}$
- $C'_- = \emptyset$
- $C_+ = \emptyset$

## 7.2 Preuves de propriétés initiales par complétion d'automate

Nous nous proposons de coder le système de déduction  $\mathcal{I}$  sous forme de termes et de règles de réécriture afin de réutiliser notre outil *Timbuk* pour rechercher une dérivation réussie.

### 7.2.1 Modélisation du système

Nous reprenons la structure de données du système précédent, à ceci près que nous ne représentons pas les hypothèses de récurrence, celles-ci seront incluses dans le système de réécriture. Nous décrivons un ensemble de termes de la forme **deriv**(**CN**, **CP**) (comme nous avons fait jusqu'ici, les variables sont toujours en majuscules).

- **CN** est une variable qui représente les conjectures négatives <sup>19</sup>

<sup>19</sup>c'est à dire un ensemble de conjecture de la forme  $t_1 \neq t_2$

– CP est une variable qui représente les conjectures positives <sup>20</sup>

CN et CP vont représenter les ensembles de conjectures sous forme de pile. Le terme **empty** représentera une pile vide. Les règles de réécriture ne pourront agir que sur la tête de ces piles.

Considérons l'ensemble de conjectures  $\{t_1 = t_2, t_3 = t_4, t_5 \neq t_6\}$ , chacune des égalités (resp. diségalités) sera représenté grâce au symbole **eq** (resp. **dis**) et l'ensemble sera représenté par le terme

$\text{deriv}(\text{cn}(\text{dis}(t_5, t_6), \text{empty}), \text{cp}(\text{eq}(t_1, t_2), \text{cp}(\text{eq}(t_3, t_4), \text{empty})))$

Supposons que nous voulions prouver une diséquation  $\mathcal{F}$ . La première étape va être d'appliquer la règle **rec** à la main, c'est à dire de remplacer  $\mathcal{F}$  par deux ensembles de conjectures : positives et négatives où toutes les variables seront skolémisées. A partir de là, nous représentons ces deux ensembles par un terme  $t$  de la forme  $\text{deriv}(\text{CN}, \text{CP})$  et nous calculons  $\mathcal{A}_t$  (voir définition 33). Nous allons étudier la complétion de cet automate par un système de réécriture comprenant :

1. la description du système d'inférence  $\mathcal{I}$
2. Les hypothèses de récurrence
3. la théorie **T**

### Codage de $\mathcal{I}$

Nous reprenons chaque règle du système d'inférence  $\mathcal{I}$  et nous les représentons par des règles de réécriture agissant sur  $\mathcal{A}_t$ . Les trois premières règles (**simplify 1, 2 et 3**) seront déjà inutiles puisque la théorie et les hypothèses de récurrence seront intégrées au système de réécriture. Nous donnons la représentation de toutes les règles utilisées :

**commutativité** Les deux premières règles codent la commutativité des opérateurs **eq** et **dis**.

$$\text{eq}(x, y) \rightarrow \text{eq}(y, x)$$

$$\text{dis}(x, y) \rightarrow \text{dis}(y, x)$$

**elim** - Cette règle elimine les diségalités commençant par deux symboles constructeurs différents

$$(\mathcal{E}, Hr, C_- \cup \{f(\vec{t}_1) \neq g(\vec{t}_2)\}, C_+) \vdash_{\mathcal{I}} (\mathcal{E}, Hr, C_-, C_+) \text{ (si } f, g \in \mathcal{C}, f \neq g)$$

Chaque couple de constructeur génère une règle, nous donnons l'exemple des naturels :

$$\text{deriv}(\text{HR}, \text{cn}(\text{dis}(s(x), 0), \text{CN}), \text{CP}) \rightarrow \text{deriv}(\text{HR}, \text{CN}, \text{CP})$$

**elim** + Cette règle élimine les tautologies dans les conjectures positives

$$(\mathcal{E}, Hr, C_-, C_+ \cup \{t = t\}) \vdash_{\mathcal{I}} (\mathcal{E}, Hr, C_-, C_+)$$

elle sera représentée par la règle de réécriture suivante :

$$\text{deriv}(\text{HR}, \text{CN}, \text{cp}(\text{eq}(x, x), \text{CP})) \rightarrow \text{deriv}(\text{HR}, \text{CN}, \text{CP})$$

**simp** + Cette règle simplifie les conjectures positives qui commencent par le même symbole de tête pourvu que ce soit un symbole constructeur.

$$(\mathcal{E}, Hr, C_-, C_+ \cup \{f(\vec{t}_1) = f(\vec{t}_2)\}) \vdash_{\mathcal{I}} (\mathcal{E}, Hr, C_-, C_+ \cup \{\vec{t}_1 = \vec{t}_2\}) \text{ (si } f \in \mathcal{C})$$

Comme précédemment, chaque symbole constructeur qui n'est pas un symbole de constante, génère une règle, nous donnons une fois de plus l'exemple des naturels

$$\text{deriv}(\text{HR}, \text{CN}, \text{cp}(\text{eq}(s(x), s(y)), \text{CP})) \rightarrow \text{deriv}(\text{HR}, \text{CN}, \text{cp}(\text{eq}(x, y), \text{CP}))$$

<sup>20</sup>c'est à dire un ensemble de conjectures de la forme  $t_1 = t_2$

## Les hypothèses de récurrence

Si la conjecture étudiée admet une variable de récurrence, alors elle génère un ensemble d'hypothèses de récurrence. Nous proposons de modéliser cet ensemble de la même manière que les conjectures : écrire un automate représentant ces hypothèses puis de le compléter, en utilisant une approximation exacte et à l'aide de la théorie et des règles **decomp** et **simplify 2** de  $\mathcal{I}$ . L'automate obtenu représente toutes les hypothèses de récurrence que nous avons le droit d'utiliser suivant  $\mathcal{I}$ . Nous traduisons les égalités obtenues sous forme de règles de réécriture et les ajoutons au système de réécriture.

## La théorie

Pour finir le système de réécriture contient toutes les règles de la théorie et éventuellement un ensemble de théorèmes inductifs.

### 7.2.2 Principe général

Maintenant que nous avons à notre disposition :

1. Un automate  $\mathcal{A}$  dont tous les états sont utiles et accessibles et reconnaissant exactement le terme  $\mathbf{deriv}(\mathbf{C-}, \mathbf{C+})$  où  $\mathbf{C-}$  et  $\mathbf{C+}$  sont les buts positifs et négatifs à prouver
2. Un système de réécriture que nous nommons  $\mathcal{R}_d$  contenant d'une part la théorie  $\mathbf{T}$ , les hypothèses de récurrence  $\mathcal{H}r$ , éventuellement un ensemble de théorèmes inductifs, et d'autre par une modélisation des règles de  $\mathcal{I}$ .

Nous allons compléter l'automate à l'aide d'une approximation  $\alpha$  exacte et chercher s'il existe un automate  $\mathcal{A}_{\mathcal{R},\alpha}^k$  reconnaissant au moins un terme de la forme  $\mathbf{deriv}(\mathbf{empty}, \mathbf{empty})$ . Si un tel terme est accessible alors nous aurons montré qu'il existe une dérivation partant de  $\mathbf{deriv}(\mathbf{C+}, \mathbf{C-})$  et menant à  $\mathbf{deriv}(\mathbf{empty}, \mathbf{empty})$ , la propriété sera montrée.

### 7.2.3 Reprenons notre exemple

Nous reprenons l'exemple développé en 7.1.4 sur la théorie décrivant les fonctions *plus* et *moins*. cette théorie s'oriente et forme de système de réécriture suivant :

$$\mathit{plus}(0, x) \rightarrow x \tag{7.17}$$

$$\mathit{plus}(s(x), y) \rightarrow s(\mathit{plus}(x, y)) \tag{7.18}$$

$$\mathit{moins}(0, x) \rightarrow 0 \tag{7.19}$$

$$\mathit{moins}(x, 0) \rightarrow x \tag{7.20}$$

$$\mathit{moins}(s(x), s(y)) \rightarrow \mathit{moins}(x, y) \tag{7.21}$$

et que nous voulons prouver la conjecture suivante

$$\forall x \forall y \mathit{plus}(x, s(y)) \neq \mathit{moins}(x, y) \tag{7.22}$$

Cette conjecture admet la variable  $x$  comme variable de récurrence.

1. La première étape consiste à générer les ensembles de conjectures à prouver, ce qui revient à appliquer à la main la règle **rec**.

$C_-$  L'ensemble des conjectures négatives à prouver est

$$\{plus(0, s(0)) \neq moins(0, 0), plus(0, s(s(\bar{y}))) \neq moins(0, s(\bar{y}))\}$$

nous renommons  $\bar{y}$  en  $a$  et cet ensemble se code par le sous-terme

$$cn(\text{dis}(plus(0, s(0)), moins(0, 0)), cn(\text{dis}(plus(0, s(s(a))), moins(0, s(a))), \text{empty}))$$

$C_+$  Nous avons

$$C_+ = \{Plus(\bar{x}, S(0)) = Moins(\bar{x}, 0), Plus(\bar{x}, S(S(\bar{y}))) = Moins(\bar{x}, S(\bar{y}))\}$$

Nous représentons cet ensemble de la même façon que  $C_-$  en renommant les variables skolémisées par  $b, c, d$ . Il ne reste plus qu'à écrire un automate  $\mathcal{A}_{+,-}$  reconnaissant  $\text{deriv}(\mathbf{CN}, \mathbf{CP})$ , cet automate est donné en annexe D.1

2. Ensuite nous écrivons un automate  $\mathcal{A}_{Hr}$  reconnaissant les règles de récurrence

$$\{plus(s(b), s(0)) = moins(s(b), 0), plus(s(c), s(s(d))) = moins(s(c), s(d))\}$$

Puis nous complétons cet automate de manière exacte à l'aide de  $\mathbf{T}$  et les règles de  $\mathcal{I}$  qui s'appliquent aux règles de récurrence, à savoir **simplify 2** et **decomp**. Ce calcul se termine et nous obtenons les équivalences suivantes :

$$\begin{array}{ll} s(plus(b, 0)) & \leftrightarrow b \\ b & \leftrightarrow plus(b, s(0)) \\ b & \leftrightarrow s(b) \\ moins(s(c), s(d)) & \leftrightarrow plus(s(c), s(s(d))) \\ moins(c, d) & \leftrightarrow plus(s(c), s(s(d))) \end{array}$$

Pour finir, à l'aide de *Timbuk*, nous effectuons la complétion de  $\mathcal{A}_{+,-}$  par l'ensemble des règles décrivant le système, les hypothèses de récurrence et les règles codant la commutativité de *plus* et le lemme 7.16. Le processus de complétion ne s'arrête pas, mais au bout de 8 étapes de complétion, le terme  $\text{deriv}(\text{empty}, \text{empty})$  est reconnu. La propriété est prouvée.

## 7.2.4 Limite de l'approche

L'algorithme que nous venons de proposer offre l'avantage d'être entièrement automatique puisque l'application de la règle **rec** est implémentable, malheureusement le calcul diverge très facilement puisque nous ne faisons pas de restriction sur la terminaison du système de réécriture utilisé. Le problème que cela engendre est que l'automate considéré grossit très vite, rendant l'utilisation de l'outil difficile. De plus il faut encore vérifier la condition de linéarité ce qui pose toujours des problèmes de complexité. C'est pourquoi nous nous proposons dans ce qui suit d'étendre notre approche aux preuves par interprétation abstraite.

## 7.3 Vers l'interprétation abstraite

Nous proposons d'étendre cet algorithme aux preuves de propriétés négatives par interprétation abstraite, ceci a été présenté dans [GVTT02]. L'intérêt d'utiliser l'interprétation abstraite est que cela va permettre de simplifier les conjectures et que si nous réussissons à les prouver à

l'aide d'une abstraction par interprétation abstraite ( $A$ ) alors il existe une preuve sans  $A$ . En effet, intuitivement, l'abstraction va *confondre* des termes distincts dans  $\mathcal{T}(\mathcal{F})/\equiv_{\mathbf{T}}$  donc si nous arrivons à prouver que deux termes sont disjoints  $\mathcal{T}(\mathcal{F})/\equiv_{\mathbf{T} \cup}$  alors ces deux termes sont encore disjoints dans  $\mathcal{T}(\mathcal{F})/\equiv_{\mathbf{T}}$ . Nous proposons un exemple dans la théorie définissant *plus* et *mult*.

$$\text{plus}(0, x) \rightarrow x \quad (7.23)$$

$$\text{plus}(s(x), y) \rightarrow s(\text{plus}(x, y)) \quad (7.24)$$

$$\text{mult}(0, y) \rightarrow 0 \quad (7.25)$$

$$\text{mult}(s(x), y) \rightarrow \text{plus}(y, \text{mult}(x, y)) \quad (7.26)$$

$$(7.27)$$

Nous cherchons à montrer que  $\forall x \text{mult}(x, x) \neq s(s(0))$ . Pour cela, nous approximons le modèle initial à l'aide des égalités suivantes :

$$\begin{aligned} s(0) &= \text{un} \\ s(s(0)) &= \text{deux} \\ s(s(s(x))) &= \text{sup2} \end{aligned}$$

Ensuite, nous reprenons la méthode proposée à la section précédente, nous écrivons un automate reconnaissant sous forme d'un terme  $\text{deriv}(\text{CN}, \text{empty})$  où  $\text{CN}$  est l'ensemble des conjectures

$$\text{mult}(0, 0) \neq \text{deux}$$

$$\text{mult}(\text{un}, \text{un}) \neq \text{deux}$$

$$\text{mult}(\text{deux}, \text{deux}) \neq \text{deux}$$

$$\text{mult}(\text{sup2}, \text{sup2}) \neq \text{deux}$$

Nous complétons l'automate par les règles du système  $\text{deriv}$ , les règles de la théorie, et les règles de l'approximation. La complétion se fait en utilisant une approximation exacte, en ajoutant toujours de nouveaux états. Nous cherchons le terme  $\text{deriv}(\text{empty}, \text{empty})$ , ce terme est reconnu par l'automate à la septième étape de complétion, la propriété est montrée.

## 7.4 Et la complétude réfutationnelle ?

Nous avons proposé un système de déduction pour la preuves de propriétés initiales, et nous avons montré que notre algorithme était correct. Malheureusement, nous n'avons pas eu le temps d'étudier la complétude réfutationnelle de cet algorithme. Nous pensons que pour obtenir la complétude réfutationnelle, il faudrait utiliser des ensembles tests à la place de ensembles couvrants [BKR95]. En supposant que la complétude réfutationnelle de notre système soit démontrable, si nous trouvons une abstraction permettant de terminer la complétion sans réussir à prouver les hypothèses, si le système utilisé est réfutationnellement complet, nous aurions montré que les conjectures étaient fausses.



8

## Bilan et perspectives

## 8.1 Bilan

Ce travail a été divisé en trois parties, la première présente la complétion d'automates et reprend des travaux de Thoams Gent et présentés dans [Gen98], la deuxième partie est une application des outils théoriques présentés précédemment à la vérification de protocoles de sécurité, enfin la dernière partie étend la réflexion aux preuves de propriétés initiales.

### 8.1.1 Extension de la complétion aux systèmes non linéaires et calcul exact

Nous avons montré comment il était possible d'utiliser la structure des automates d'arbres pour calculer, grâce à une complétion à la Knuth-Bendix, une sur-approximation des termes atteignables à partir d'un langage régulier pour un système de réécriture donné.

Notre première contribution est algorithmique, nous avons en effet proposé un algorithme qui améliore la recherche des paires critiques nécessaire lors de la complétion. Nous avons prouvé que notre algorithme était correct et complet puis nous l'avons intégré à *Timbuk*. Notre implémentation a amélioré les performances de l'outil pour la complétion d'automates en divisant par six le temps de calcul moyen, ce qui nous a permis de compléter des automates allant jusqu'à plus de 700 transitions.

Parallèlement, nous avons étendu la complétion d'automates aux systèmes de réécriture non linéaires à gauche en introduisant la notion de condition de linéarité. Nous avons montré que pour un système de réécriture non linéaire à gauche, et pour tout automate  $\mathcal{A}$ , vérifiant la condition de linéarité induite par  $\mathcal{R}$ , si nous trouvons une fonction d'abstraction permettant de réussir la complétion alors le langage reconnu par l'automate complet contient l'ensemble des termes atteignables par  $\mathcal{R}$  à partir de  $\mathcal{A}$ .

Enfin, nous avons donné une fonction d'abstraction  $\alpha_0$  pour laquelle, si la complétion d'un automate  $\mathcal{A}$  par un système de réécriture  $\mathcal{R}$ , linéaire ou vérifiant la condition de linéarité, termine alors c'est un calcul exact des descendants.

### 8.1.2 Application à la vérification de protocoles cryptographiques

Reprenant le travail que de Thomas Genet et Francis Klay proposé dans [GK00], nous avons appliqué la complétion d'automates pour vérifier des protocoles cryptographiques. Cette méthode utilise des langages réguliers et des règles de réécriture pour modéliser des protocoles. Les propriétés sont alors représentées par des termes interdits, la propriété est vérifiée lorsque ces termes sont inatteignables. Nous avons utilisé cette approche pour modéliser puis vérifier un protocole de protection du contenu numérique développé par *Thomson Multimédia* : le protocole *View Only* de **SmartRight**. Cette étude nous a permis de montrer que la représentation par automates était particulièrement bien adaptée pour vérifier des propriétés *atypiques* comme c'était le cas de la propriété d'*Anti Replay* de ce protocole.

### 8.1.3 Un système de preuve de propriétés initiales

Nous avons donc utilisé la complétion d'automates pour savoir si un ensemble régulier de termes est atteignable par un système de réécriture  $\mathcal{R}$  et un langage régulier (décrit par un automate) donné; ce qui revient à étudier des propriétés de la forme

$$t_1 \not\equiv_{\mathbf{T}} t_2$$

pour tout couple de termes clos  $(t_1, t_2)$  appartenant à des langages réguliers. Nous avons ensuite étendu notre travail aux propriétés de même forme où cette fois-ci les termes  $t_1$  et  $t_2$  sont susceptibles de contenir des variables. Ce qui revient à prouver des propriétés initiales c'est à dire des propriétés vraies dans le plus petit modèle de Herbrand de  $\mathbf{T}$ . Nous avons proposé un premier système d'inférence utilisant des règles de déduction dont la plus importante est la règle d'application du raisonnement par descente infinie. Pour terminer, nous avons utilisé la complétion d'automate pour implémenter ce système d'inférence, ramenant ainsi la preuve de propriétés initiales à la recherche d'un ensemble de termes atteignables.

## 8.2 Perspectives

Il est nécessaire d'améliorer notre outil de vérification afin de le rendre plus accessible. Un problème de notre méthode de vérification est qu'elle suppose que l'utilisateur cherchant à étudier un protocole soit capable :

- de décrire les configurations de son protocole par un automate,
- de donner une fonction d'abstraction pour réussir la complétion de l'automate.

Afin d'améliorer notre approche, il nous semble très utile de pouvoir proposer une traduction la plus automatique possible d'un protocole partant de sa spécification sous forme usuelle.

Parallèlement, nous envisageons de faire une interface de *Timbuk* permettant de manipuler graphiquement les automates, ce qui faciliterait grandement la recherche de la fonction d'abstraction. L'idéal serait d'avoir une idée graphique des transitions créées lors d'une étape de complétion, ce qui permettrait de savoir si elles sont similaires à des transitions ajoutées lors de précédentes étapes.

Par ailleurs, il serait intéressant d'étudier d'autres protocoles cryptographiques présentant de nouvelles propriétés à modéliser et vérifier.

Enfin, il reste encore beaucoup de travail à faire sur le système de preuve de propriétés initiales proposé :

- l'étude de la complétude réfutationnelle,
- l'étude de l'intégration des propriétés à plusieurs variables de récurrence.

Enfin, nous souhaiterions utiliser cette approche pour compléter l'étude des protocoles cryptographiques.

Ce qui m'a intéressé dans ce travail a été de développer une partie théorique et de pouvoir l'appliquer à des problèmes pratiques, c'est d'ailleurs ce que j'aimerais faire en améliorant le système de déduction proposé en dernier lieu.



A

# Implémentation de l'algorithme de filtrage

## A.1 Construction de l'automate d'un terme

Nous détaillons ici l'implémentation de l'algorithme utilisé dans TIMBUK.

$\mathcal{A} = \langle \mathcal{F}, \mathcal{Q}, \mathcal{Q}_f, \Delta \rangle$  un automate,  $\mathcal{R}$  un système de réécriture, nous cherchons l'ensemble des substitutions  $\sigma$  telles qu'il existe un terme  $t$  reconnu par  $\mathcal{A}$  et un membre gauche  $l$  d'une règle  $l \rightarrow r$  de  $\mathcal{R}$  vérifiant  $t = l\sigma$ .

### CONSTRUCTION D'UN AUTOMATE RECONNAISSANT $\mathcal{R}$

Nous allons construire un automate  $\mathcal{A}_{\mathcal{R}}$  tel que si  $l$  est un membre gauche d'une règle de  $\mathcal{R}$  alors il existe un état final  $q$  de  $\mathcal{A}_{\mathcal{R}}$  tel que  $q$  reconnaît  $l$  et seulement  $l$ .

Pour chaque membre gauche  $l$  de règle de  $\mathcal{R}$ , nous construisons l'automate  $\mathcal{A}_l = \langle \mathcal{F}, \mathcal{Q}_l, \mathcal{Q}_{f_l}, \Delta_l \rangle$  du term  $l$ . Nous choisissons les ensemble d'état de sorte que les automates de membres gauche soient disjoint deux à deux.

```
(* Construction of {\em variable states} from variables *)
let make_special_state_from_variable x :state =
  Special(Fsym(var_symb, [Var(x)]))

(* new states used for normalisation of terms *)
let make_new_state () :state=
  begin
    comp_state:=!comp_state+1;
    Const(Symbol.of_string("##q" ^ string_of_int(!comp_state)));
  end

(* Construction of transition table of a term-automata *)
let make_transition_table l :transition_table =
  let list_assoc = ref l in
  let find_and_delete x =
    match
      x
    with
      Var(_) -> List.assoc x !list_assoc
    | _ ->
      let res = List.assoc x !list_assoc
      in list_assoc := List.remove_assoc x !list_assoc;res
  in
  let rec aux l =
    match
      l
    with
      [] -> []
    | (t,q)::ll ->
      match
        t

```

```

with
  Var(x) -> (aux ll)
| Const (a) ->
  [new_trans (Symbol.of_string (Term.to_string t)) [] q]
  @(aux ll )
| Fsym(f,list) ->
  [new_trans f (flat_map (fun x ->[find_and_delete x])list)q]
  @(aux ll)
| _ ->
  failwith "Term should not be special
-- error in make_transition_table"
  in Transition.list_to_trs (aux l)

(*create an automaton whose langage is empty *)
let term_automaton_empty =
  Automaton.make_automaton
    Alphabet.new_alphabet
    Alphabet.new_alphabet
    State_set.empty
    State_set.empty
    (Transition.list_to_trs [])
    (Transition.list_to_trs [])

(* translate a left member of a rewriting system into a term automaton *)
let translate_one_term_into_automaton t i alphabet =
  let abstract =
    match
      t
    with
      Var(x) -> [(t,(make_special_state_from_variable x))]
    | Const(a) -> [(t,Fsym(Symbol.of_string("rule_"^string_of_int(i)),[]))]
    | Fsym(f,l) -> (
      let set_subterm = Common.flat_map set_of_sub_term l
      in
      [(t,Fsym(Symbol.of_string("rule_"^string_of_int(i)),[]))]
      @(make_abstraction_table set_subterm))
    | _ -> failwith "error in translate_one_term_into_automaton
-- term should not be Special"
  in
  let transition= make_transition_table abstract
  and states=(State_set.list_to_set (snd(List.split abstract)))
  and final_states =(State_set.list_to_set (List.find_all
    is_final_state (snd(List.split abstract))))
  in
  Automaton.make_automaton
    alphabet

```

```

    (make_elt_alphabet var_symb 1)
    states
    final_states
    (make_transition_table [])
    transition

(* for a given alphabet \al\ and list of term \list_term\,
\translator al list_term\ is an automaton which recognized \list_term\ *)
let translator alphabet list_term =
  let rec aux alphabet list_term i=
    match
      list_term
    with
      [] -> term_automaton_empty
    | t::tail -> Automaton.make_fast_union
      (translate_one_term_into_automaton t i alphabet ) (aux
        alphabet tail (i+1))
  in aux alphabet list_term 1

(* for a given alphabet rewriting system \rws\ and an alphabet \al\,
\translator_lhs rws al\ is our automaton \A_{\mathcal{R}}\ *)
let translate_lhs (rws:TRS.ruleSystem) (alphabet: alphabet)=
  let res = translator alphabet (list_term_lhs rws) in
  print_string (Automaton.to_string res); res

```

## A.2 calcul des transitions utiles

Nous calculons tout d'abord  $\mathcal{A}_\cap = \mathcal{A} \cap \mathcal{A}_{\mathcal{R}}$ .

```

(* Specific automaton intersection for [term automaton] inter [usual automaton].
BE-CAREFUL: For efficiency reasons, the state set field is not computed...
but the final state set is. *)

let intersection term_automaton usual_automaton =
  let at= Automaton.inter term_automaton usual_automaton in
  let list_states= list_states_of_transitions (Automaton.get_transitions at) in
  let final_states= Common.my_find_all is_final_state list_states in
  Automaton.make_automaton
    (Automaton.get_alphabet at)
    (Automaton.get_state_ops at)
    (State_set.empty)
    (State_set.list_to_set final_states)
    (Automaton.get_prior at)
    (Automaton.get_transitions at)

```

Pour des raisons d'efficacité, nous ne gardons de la table de transition de  $\mathcal{A} \cap \mathcal{A}_{\mathcal{R}}$  que les transitions *utiles* :

un état est dit *utile* s'il apparaît dans une chaîne de réécriture menant à un état final. Une

transition sera dite *utile* si elle contient au moins un état *utile*.

Dans le code qui suit  $at_{old}$  représente l'automate  $\mathcal{A}_{\mathcal{R},\alpha}^n \cap \mathcal{A}_{\mathcal{R}}$  et  $at_{new}$  représente l'automate complément  $\mathcal{A}_{\mathcal{R},\alpha}^{n+1} \cap \mathcal{A}_{\mathcal{R}} \setminus \mathcal{A}_{\mathcal{R},\alpha}^n \cap \mathcal{A}_{\mathcal{R}}$ ,  $hsh_{all}$  et  $hsh_{new}$  représente simplement les transitions de  $at_{old}$  et  $at_{new}$  sous forme de tables de hachage.

```

let find_useful_transition at_old at_new hsh_all hsh_new=
  let vu=ref [] and list_states=ref [] in
  let new_transition= TRS.to_list(Automaton.get_transitions at_new) in
  let all_transition= new_transition
    @(TRS.to_list(Automaton.get_transitions at_old))

  in
  let appearing_in tr = (* on vire les special*)
  List.map (fun x ->
    match
      x
    with
      Special y -> y
      | _ ->x) (get_non_substitution_part (Automaton.lhs tr))
  in
  let right_member tr =
    (fun x ->
      match
        x
      with
        Special y -> y
        | _ ->x)
    (Automaton.rhs tr) (*pas special*)
  in
  let trans_that_leads_to q hsh_all = (*q pas special*)
    try (Hashtbl.find
      (hsh_all) (q)) with Not_found -> []
  in
  let trans_where_appear q all_trans= (* pas special*)
    try Common.my_find_all
      (fun tr -> List.mem (q) (appearing_in tr))
      all_trans with Not_found -> []
  in
  let deja_vu tr = List.mem tr !vu
  in
  let rec get_useful_transition list_tr all_trans hsh_result hsh_all
    hsh_new=
    match
      list_tr
    with
      [] -> hsh_result
      | tr::llist -> (
        list_states:=(Automaton.rhs tr)::(!list_states);

```

```

    if
      deja_vu tr
    then
      get_useful_transition
      llist
      all_trans
      hsh_result
      hsh_all
      hsh_new
    else (
      vu:= tr::(!vu);
      let more_transition=(trans_where_appear (right_member
        tr) all_trans)@(Common.flat_map (fun q ->
        trans_that_leads_to q hsh_all)
        (appearing_in tr))
      in
      get_useful_transition
      (llist@more_transition)
      all_trans
      (hashtable_modif hsh_result ([tr])) hsh_all hsh_new))
  in
  let rec finish new_trans acc hsh_all hsh_new=
    match
      new_trans
    with
      [] -> acc
      | tr::list -> (finish list (get_useful_transition [tr]
        all_transition acc hsh_all hsh_new) hsh_all hsh_new )
  in
  let result = finish new_transition hsh_new hsh_all hsh_new
  in
  (result,Common.clean (!list_states))

```

### A.3 calcul des $Q$ -substitutions

Nous allons chercher à calculer l'ensemble des substitutions  $\sigma$  telles qu'il existe un terme  $t$  reconnu par  $\mathcal{A}$  et un membre gauche  $l$  d'une règle de  $\mathcal{R}$  vérifiant  $t = l\sigma$ . Pour chaque état final apparaissant dans l'ensemble des transitions utiles, nous construisons l'ensemble des substitutions qu'il définit :

```

(*)
from a config in the form f((x,q1), (q2,q3), (y,q4) )
give the part which defined a substitution
ie x <- q1, y <- q4 (the result is a substitution)
*)

```

```

let get_substitution_part (c:( 'a, 'b) Common.term_const) :substitution =
  let rec aux l acc=
    match
      l
    with
      [] -> acc
    | q::ll ->
      match
        q
      with
        Special(Fsym(prod_sym,[Special(Fsym(var_sym,[Var(y))]);state]))
        ->
          aux ll ([(y,Special(state))]@acc)
        | (Fsym(prod_sym,[Special(Fsym(var_sym,[Var(y))]);state]))
        -> aux ll ([(y,Special(state))]@acc)
        | _ -> aux ll acc
    in
    match
      c
    with
      Fsym(f,l) -> aux l []
    | _ -> []

(* from a config in the form f((x,q1), (q2,q3), (y,q4) ) give the part which doesn't
   defined a substitution ie (q2,q3) *)

let get_non_substitution_part (c:( 'a, 'b) Common.term_const) =
  let rec aux l acc =
    match
      l
    with
      [] -> acc
    | q::ll ->
      match
        q
      with
        Special(Fsym(prod_sym,[Special(Fsym(var_sym,[y])]);state])) -> (aux ll acc)
        | Special(Fsym(prod_sym,[q1;q2])) -> (aux ll (acc@[q]))
        | (Fsym(prod_sym,[Special(Fsym(var_sym,[y])]);state])) -> (aux ll acc)
        | (Fsym(prod_sym,[q1;q2])) -> (aux ll (acc@[q]))
        | _ -> failwith "not an intersection automata"
    in
    match

```

```

    c
  with
    Fsym(f,l) -> aux l []
    | _ -> []

let rec mixe_return list_of_list =
  match
    list_of_list
  with
    |list1::list2:: tail ->
      let rec distribute list return =
        match list with
          | h :: t -> distribute t ((Common.my_map (fun l -> h @ l) list2) @ return)
          | [] -> return
        in
          let new_list2 = distribute list1 [] in
            mixe_return (new_list2 :: tail)
      | list :: [] -> list
      | [] -> [[]]

let rec distrib (subst:substitution) subst_list=
  List.map (fun x -> subst@x) subst_list

let get_subst (q:state) hash_df: substitution list=
  let term_list_recognized_by q (* liste de termes *) =
    try Common.my_map Automaton.lhs (Hashtbl.find hash_df q)
    with Not_found -> []
  in
    let subst_defined_by config (* list de substitution ((x,q1);(y,q2))*) =
      get_substitution_part config
    and leaving_states_in config (*liste d'états *) =
      get_non_substitution_part config
    in
      let rec aux (q:state)=
        let qq= (match q with Special(y) -> y | _ -> q) in
          let subst_list_defined_by_leaving_states config=
            let rec cut_empty_subst_branches l res_substs=
              match l with
                [] -> res_substs
                | q::rem ->
                  let lsubst= aux q in
                    if lsubst=[]
                      (* if one of the partial substitution list is empty there is no solution *)

```

```

        then [[]]
        (* the no-solution substitution of mixe_return! @&!&@ *)
        (* otherwise the substitution is conserved and added to the list *)
        else (cut_empty_subst_branches rem (lsubst::res_substs))
    in
        cut_empty_subst_branches (leaving_states_in config) []
(*
    List.map (fun x -> aux x) (leaving_states_in config) *)

in
let subst= try (Hashtbl.find !subst_by_state qq)
with

    (* if the substitution has not been already computed, we do it *)
    Not_found ->
    let new_subst=
    Common.flat_map
    (fun config ->((distrib(subst_defined_by config)
    (mixe_return (subst_list_defined_by_leaving_states config))))))
    (term_list_recognized_by qq)
    in
        let cleant= (Common.clean (clean_lsubst new_subst)) in
        let _= Hashtbl.add !subst_by_state qq cleant in
        subst_by_state:=!subst_by_state ;
        new_subst
    in subst
in
    aux q

```



B

## Fonctions d'abstraction

## B.1 Fonction d'abstraction d'un polynôme

Nous proposons ici la fonction d'abstraction utilisé lors de la complétion de l'automate reconnaissant le polynôme  $P(x) = x^6 - 2x^5 - 2x^4 + 2x^3 + x^2 - 1$ . Cette abstraction est décrite sous formes de règles de normalisation que nous donnons dans la syntaxe *Timbuk*.

Approximation Poly5

States

```

q0 q1 q2 q3 q4 q5 q6 q7 q8 q9
q10 q11 q12 q13 q14 q15 q16 q17 q18 q19
q20 q21 q22 q23 q24 q25 q26 q27 q28 q29
q30 q31 q32 q33 q34 q35 q36 q37 q38 q39
q40 q41 q42 q43 q44 q45 q46 q47 q48 q49
q50 q51 q52 q53 q54 q55 q56 q57 q58 q59
qf
qneg

```

Rules

```

[x -> y] -> [Mult(z,q0) -> q0]
[x -> y] -> [Mult(q0,z) -> q0]
[x -> y] -> [Mult(q2,z) -> q3]
[x -> y] -> [Mult(z,q2) -> q3]
[x -> y] -> [Mult(z,q3) -> q3]
[x -> y] -> [Mult(q3,z) -> q3]
[x -> y] -> [Mult(q1,q22) -> q22]
[x -> y] -> [Mult(q1,q27) -> q27]
[x -> y] -> [Mult(q1,q26) -> q26]
[x -> y] -> [Mult(q1,q25) -> q25]
[x -> y] -> [Mult(q1,q24) -> q24]
[x -> y] -> [Mult(q1,q18) -> q18]
[x -> y] -> [Mult(q1,q19) -> q19]
[x -> y] -> [Mult(q1,q16) -> q16]
[x -> y] -> [Mult(q1,q10) -> q10]
[x -> y] -> [Mult(q1,q11) -> q11]
[x -> y] -> [Mult(q1,q8) -> q8]
[x -> y] -> [Mult(q1,q6) -> q6]
[x -> y] -> [Mult(q1,q23) -> q23]
[x -> y] -> [Mult(q1,q7) -> q7]
[x -> y] -> [Mult(qneg,z) -> qneg]

[x -> y] -> [Plus(q0,q0) -> q0]
[x -> y] -> [Plus(q0,q9) -> q9]
[x -> y] -> [Plus(q0,q13) -> q13]
[x -> y] -> [Plus(q0,q37) -> q37]
[x -> y] -> [Plus(q1,q0) -> q1]
[x -> y] -> [Plus(q0,q17) -> q1]
[x -> y] -> [Plus(q0,q21) -> q1]

```

[x -> y] -> [Plus(qneg,q7) -> qneg]  
[x -> y] -> [Plus(qneg,q13) -> qneg]  
[x -> y] -> [Plus(qneg,z) -> qneg]  
[x -> y] -> [Plus(z,t) -> q3]

[x -> y] -> [Moins(q0,q0) -> q0]  
[x -> y] -> [Moins(q3,z) -> q3]  
[x -> y] -> [Moins(z,t) -> qneg]

## B.2 Règles d'abstraction pour *View Only* sans intrus

Les règles suivantes sont des règles de normalisations qui définissent la fonction d'abstraction utilisée pour la complétion de l'automate *init* représentant une configuration initiale sans intrus du protocole *View Only*. Ces règles sont données dans la syntaxe *Timbuk*

Approximation Proto

States

```

qlistold qlistcurrent qlistnext qnil qmsg1 qmsg2 qmsg3 qmsg4
qcc1 qtc1 qcc2 qtc2 qcc3 qtc3 qcc4 qtc4 qstore1 qstore2 qstore3
qkc qkcc qold qcurrent qnext qreadold qreadcurrent qreadnext qvokeyold
qvokeycurrent qvokeynext qvoriold qvoricurrent qkvoriold qkvoricurrent
qkvorinext qvorioldcurrent qvorinext qvorold qvorcurrent qvornext qkvorold
qkvorcurrent qkvornext qrun qconsnext1 qconsold1 qconscurrent1 qconsnext2
qconsold2 qconscurrent2 qvokeyseedold qvokeyseedcurrent
qvokeyseednext

```

Rules

```

[x -> y] -> [Kc -> qkcc
  Key(qkcc) -> qkc
  run -> qrun
  read(qold, z) -> qreadold
  read(qcurrent, z) -> qreadcurrent
  read(qnext, z) -> qreadnext

  vokey(qold) -> qvokeyseedold
  vokey(qcurrent) -> qvokeyseedcurrent
  vokey(qnext) -> qvokeyseednext
  Key(qvokeyseedold) -> qvokeyold
  Key(qvokeyseedcurrent) -> qvokeycurrent
  Key(qvokeyseednext) -> qvokeynext

  vori(qvokeyold, qnil) -> qvoriold
  vori(qvokeyold, qreadold) -> qvoriold
  vori(qvokeycurrent, qnil) -> qvoricurrent
  vori(qvokeycurrent, qreadold) -> qvoricurrent
  vori(z, qreadcurrent) -> qvorinext
  vori(z, qreadnext) -> qvorinext
  vori(qvokeynext, z) -> qvorinext
  Key(qvoriold) -> qkvoriold
  Key(qvoricurrent) -> qkvoricurrent
  Key(qvorinext) -> qkvorinext

  vor(qold) -> qkvorold
  vor(qcurrent) -> qkvorcurrent
  vor(qnext) -> qkvornext

```

```

Key(qkvorold) -> qvorold
Key(qkvorcurrent) -> qvorcurrent
Key(qkvornext) -> qvornext
ConverterCard(qlistold, u, v) -> qcc1
ConverterCard(qlistcurrent, u, v) -> qcc2
ConverterCard(qlistnext, u, v) -> qcc3
ConverterCard(qnil, u, v) -> qcc4
cons(qvokeyold, z) -> qconsold1
cons(qvokeycurrent, z) -> qconscurrent1
cons(qvokeynext, z) -> qconsnext1
]

```

```

[x -> qconsold1] -> [z -> qconsold1]
[x -> qconscurrent1] -> [z -> qconscurrent1]
[x -> qconsnext1] -> [z -> qconsnext1]
[encrypt(qvokeyold, x) -> y] -> [z -> qconsold2]
[encrypt(qvokeycurrent, x) -> y] -> [z -> qconscurrent2]
[encrypt(qvokeynext, x) -> y] -> [z -> qconsnext2]

```

```

[State(x, qcc1, z, u) -> v] -> [x -> qstore1 u -> qstore1]
[State(x, qcc2, z, u) -> v] -> [x -> qstore2 u -> qstore2]
[State(x, qcc3, z, u) -> v] -> [x -> qstore3 u -> qstore3]
[State(x, qcc4, z, u) -> v] -> [x -> qstore3 u -> qstore3]

```

```

[store(x, y) -> z] -> [x -> z y -> z]
[cons(x, y) -> qstore1] -> [x -> qstore1 y -> qstore1]
[cons(x, y) -> qstore2] -> [x -> qstore2 y -> qstore2]
[cons(x, y) -> qstore3] -> [x -> qstore3 y -> qstore3]

```



C

Automate complet pour SmartRight  
sans intrus

## C.1 Automate complet pour *View Only*

L'automate suivant est obtenu par *Timbuk* par complétion de l'automate *init* donné en 5.1.3 par le système  $\mathcal{R}_{proto}$

**Automaton** VO\_complet

**States**

*qnew42 qnew41 qnew40 qnew39 qnew38 qnew37 qnew36 qnew35 qnew34*  
*qnew33 qnew32 qnew31 qnew30 qnew29 qnew28 qnew27 qnew26 qnew25*  
*qnew24 qnew23 qnew22 qnew21 qnew20 qnew19 qnew18 qnew17 qnew16*  
*qnew15 qnew14 qnew13 qnew12 qnew11 qnew10 qnew9 qnew8 qnew7*  
*qnew6 qnew5 qnew4 qnew3 qnew2 qnew1 qnew0 qmsg1 qmsg2 qmsg3*  
*qmsg4 qcc2 qtc2 qcc3 qtc3 qcc4 qtc4 qstore2 qstore3 qkc qkkc*  
*qreadold qreadcurrent qreadnext qvokeyold qvokeycurrent qvokeynext*  
*qvoriold qkvoriold qkvorioldcurrent qkvorinext qvorioldcurrent qvorinext*  
*qvorold qvorcurrent qvornext qkvorold qkvorcurrent qkvornext qrun*  
*qconsnext1 qconsold1 qconcurrent1 qconsnext2 qconsold2 qconcurrent2*  
*qvokeyseedold qvokeyseedcurrent qvokeyseednext qf qcc1 qtc1 qstore1*  
*qinit qnil qold qnext qcurrent qlistnext qlistcurrent qlistold*  
*qvoriold qoldcw qcurrentcw qnextcw qkki qki*

**Final States**

*qf*

**Transitions**

*State(qnil, qcc1, qtc1, qstore1) → qf*  
*emptystore → qstore1*  
*nil → qnil*  
*old<sub>cw</sub> → qoldcw*  
*current<sub>cw</sub> → qcurrentcw*  
*next<sub>cw</sub> → qnextcw*  
*Key(qoldcw) → qold*  
*Key(qcurrentcw) → qcurrent*  
*Key(qnextcw) → qnext*  
*cons(qnext, qlistnext) → qlistnext*  
*cons(qcurrent, qlistnext) → qlistcurrent*  
*cons(qold, qlistcurrent) → qlistold*  
*cons(qold, qlistold) → qlistold*  
*vori0 → qvoriold*  
*ConverterCard(qlistold, qnil, qnil) → qcc1*  
*TerminalCard(qnil, qvoriold, qnil) → qtc1*  
*vor(qold) → qkvorold*  
*Key(qkvorold) → qvorold*  
*vokey(qold) → qvokeyseedold*  
*Key(qvokeyseedold) → qvokeyold*

---

```

                                nil → qconsold1
                                Xor(qold, qvorold) → qconsold1
                                cons(qconsold1, qconsold1) → qconsold1
                                cons(qvokeyold, qconsold1) → qconsold1
                                Kc → qkkc
                                Key(qkkc) → qkc
ConverterCard(qlistold, qvokeyold, qvorold) → qcc1
                                encrypt(qkc, qconsold1) → qstore1
                                State(qstore1, qcc1, qtc1, qstore1) → qf
                                vor(qvokeyold, qnil) → qvoriold
                                Key(qvoriold) → qkvoriold
                                State(qkvoriold, qcc1, qnew6, qstore1) → qf
TerminalCard(qconsold1, qkvoriold, qnil) → qnew6
                                State(qstore1, qcc1, qnew4, qstore1) → qf
TerminalCard(qnew5, qstore1, qnil) → qnew4
                                cons(qconsold1, qconsold1) → qnew5
                                Key(qnew3) → qstore1
                                vor(qconsold1, qnil) → qnew3
                                State(qstore1, qcc1, qnew0, qstore1) → qf
TerminalCard(qnew1, qstore1, qnil) → qnew0
                                cons(qconsold1, qnew2) → qnew1
                                cons(qconsold1, qnil) → qnew2
                                State(qstore2, qcc2, qtc1, qstore1) → qf
                                State(qstore2, qcc2, qnew0, qstore1) → qf
                                Key(qnew3) → qconsold2
                                Hach(qconsold2) → qconsold2
                                State(qstore2, qcc2, qnew4, qstore1) → qf
                                nil → qstore1
                                encrypt(qvokeyold, qconsold2) → qstore1
                                cons(qstore1, qstore1) → qstore1
                                cons(qvorold, qstore1) → qstore1
                                State(qstore1, qcc1, qnew6, qstore1) → qf
ConverterCard(qlistcurrent, qnil, qnil) → qcc2
                                nil → qstore2
                                Hach(qkvoriold) → qconsold2
                                encrypt(qvokeyold, qconsold2) → qstore2
                                cons(qstore2, qstore2) → qstore2
                                cons(qvorold, qstore2) → qstore2
                                State(qstore2, qcc2, qnew6, qstore1) → qf
                                vor(qcurrent) → qkvorcurrent
                                Key(qkvorcurrent) → qvorcurrent
                                vokey(qcurrent) → qvokeyseedcurrent
                                Key(qvokeyseedcurrent) → qvokeycurrent
                                nil → qconscurrent1

```

*Xor*(*qcurrent*, *qvorcurrent*) → *qconcurrent1*  
*cons*(*qconcurrent1*, *qconcurrent1*) → *qconcurrent1*  
*cons*(*qvokeycurrent*, *qconcurrent1*) → *qconcurrent1*  
*ConverterCard*(*qlistcurrent*, *qvokeycurrent*, *qvorcurrent*) → *qcc2*  
*encrypt*(*qkc*, *qconcurrent1*) → *qstore2*  
*State*(*qstore2*, *qcc2*, *qnew8*, *qstore1*) → *qf*  
*TerminalCard*(*qstore2*, *qstore2*, *qreadold*) → *qnew8*  
*read*(*qold*, *qnil*) → *qreadold*  
*State*(*qstore1*, *qcc1*, *qnew7*, *qstore1*) → *qf*  
*TerminalCard*(*qstore1*, *qstore1*, *qreadold*) → *qnew7*  
*vori*(*qvokeyold*, *qreadold*) → *qvoriold*  
*State*(*qkvoriold*, *qcc1*, *qnew23*, *qstore1*) → *qf*  
*TerminalCard*(*qconsold1*, *qkvoriold*, *qreadold*) → *qnew23*  
*State*(*qstore1*, *qcc1*, *qnew22*, *qstore1*) → *qf*  
*TerminalCard*(*qnew5*, *qstore1*, *qreadold*) → *qnew22*  
*Key*(*qnew21*) → *qstore1*  
*vori*(*qconsold1*, *qreadold*) → *qnew21*  
*State*(*qstore1*, *qcc1*, *qnew20*, *qstore1*) → *qf*  
*TerminalCard*(*qnew1*, *qstore1*, *qreadold*) → *qnew20*  
*vori*(*qvokeycurrent*, *qnil*) → *qvoricurrent*  
*State*(*qkvoricurrent*, *qcc2*, *qnew19*, *qstore1*) → *qf*  
*TerminalCard*(*qconcurrent1*, *qkvoricurrent*, *qnil*) → *qnew19*  
*State*(*qstore2*, *qcc2*, *qnew18*, *qstore1*) → *qf*  
*TerminalCard*(*qnew14*, *qstore2*, *qnil*) → *qnew18*  
*Key*(*qnew17*) → *qstore2*  
*vori*(*qconcurrent1*, *qnil*) → *qnew17*  
*State*(*qstore2*, *qcc2*, *qnew16*, *qstore1*) → *qf*  
*TerminalCard*(*qnew10*, *qstore2*, *qnil*) → *qnew16*  
*vori*(*qvokeycurrent*, *qreadold*) → *qvoricurrent*  
*Key*(*qvoricurrent*) → *qkvoricurrent*  
*State*(*qkvoricurrent*, *qcc2*, *qnew15*, *qstore1*) → *qf*  
*TerminalCard*(*qconcurrent1*, *qkvoricurrent*, *qreadold*) → *qnew15*  
*State*(*qstore2*, *qcc2*, *qnew13*, *qstore1*) → *qf*  
*TerminalCard*(*qnew14*, *qstore2*, *qreadold*) → *qnew13*  
*cons*(*qconcurrent1*, *qconcurrent1*) → *qnew14*  
*Key*(*qnew12*) → *qstore2*  
*vori*(*qconcurrent1*, *qreadold*) → *qnew12*  
*State*(*qstore2*, *qcc2*, *qnew9*, *qstore1*) → *qf*  
*TerminalCard*(*qnew10*, *qstore2*, *qreadold*) → *qnew9*  
*cons*(*qconcurrent1*, *qnew11*) → *qnew10*  
*cons*(*qconcurrent1*, *qnil*) → *qnew11*

---

```

    State(qstore2, qcc2, qnew7, qstore1) → qf
    State(qstore2, qcc2, qnew15, qstore1) → qf
    State(qstore2, qcc2, qnew19, qstore1) → qf
    State(qstore1, qcc1, qnew23, qstore1) → qf
    State(qstore2, qcc2, qnew23, qstore1) → qf
    State(qstore2, qcc2, qnew22, qstore1) → qf
    State(qstore3, qcc3, qnew19, qstore1) → qf
    State(qstore3, qcc3, qnew8, qstore1) → qf
    State(qstore3, qcc3, qnew0, qstore1) → qf
    State(qstore3, qcc3, qnew6, qstore1) → qf
    State(qstore3, qcc3, qnew16, qstore1) → qf
    State(qstore3, qcc3, qnew9, qstore1) → qf
    State(qstore3, qcc3, qnew7, qstore1) → qf
    State(qstore3, qcc3, qnew13, qstore1) → qf
    State(qstore3, qcc3, qnew18, qstore1) → qf
    State(qstore3, qcc3, qnew4, qstore1) → qf
    Key(qnew17) → qconcurrent2
    Key(qnew12) → qconcurrent2
    Hach(qconcurrent2) → qconcurrent2
    State(qstore3, qcc3, qtc1, qstore1) → qf
    ConverterCard(qlistnext, qnil, qnil) → qcc3
    nil → qstore3
    Hach(qkvoricurrent) → qconcurrent2
    encrypt(qvokeycurrent, qconcurrent2) → qstore3
    cons(qstore3, qstore3) → qstore3
    cons(qvorcurrent, qstore3) → qstore3
    State(qstore3, qcc3, qnew15, qstore1) → qf
    State(qstore2, qcc2, qnew20, qstore1) → qf
    Key(qnew21) → qconsold2
    vor(qnext) → qkvornext
    Key(qkvornext) → qvornext
    vokey(qnext) → qvokeyseednext
    Key(qvokeyseednext) → qvokeynext
    nil → qconsnext1
    Xor(qnext, qvornext) → qconsnext1
    cons(qconsnext1, qconsnext1) → qconsnext1
    cons(qvokeynext, qconsnext1) → qconsnext1
    ConverterCard(qlistnext, qvokeynext, qvornext) → qcc3
    encrypt(qkc, qconsnext1) → qstore3

```

*State*(*qstore3, qcc3, qnew22, qstore1*) → *qf*  
*State*(*qstore3, qcc3, qnew20, qstore1*) → *qf*  
*State*(*qstore3, qcc3, qnew23, qstore1*) → *qf*  
     *read*(*qcurrent, qnil*) → *qreadcurrent*  
     *read*(*qcurrent, qreadold*) → *qreadcurrent*  
*State*(*qstore3, qcc3, qnew24, qstore1*) → *qf*  
*TerminalCard*(*qstore3, qstore3, qreadcurrent*) → *qnew24*  
     *read*(*qold, qreadold*) → *qreadold*  
     *vori*(*qvokeynext, qnil*) → *qvorinext*  
*State*(*qkvorinext, qcc3, qnew38, qstore1*) → *qf*  
*TerminalCard*(*qconsnext1, qkvorinext, qnil*) → *qnew38*  
     *State*(*qstore3, qcc3, qnew37, qstore1*) → *qf*  
*TerminalCard*(*qnew29, qstore3, qnil*) → *qnew37*  
     *Key*(*qnew36*) → *qstore3*  
     *vori*(*qconsnext1, qnil*) → *qnew36*  
     *State*(*qstore3, qcc3, qnew35, qstore1*) → *qf*  
*TerminalCard*(*qnew26, qstore3, qnil*) → *qnew35*  
     *vori*(*qvokeynext, qreadold*) → *qvorinext*  
*State*(*qkvorinext, qcc3, qnew34, qstore1*) → *qf*  
*TerminalCard*(*qconsnext1, qkvorinext, qreadold*) → *qnew34*  
     *State*(*qstore3, qcc3, qnew33, qstore1*) → *qf*  
*TerminalCard*(*qnew29, qstore3, qreadold*) → *qnew33*  
     *Key*(*qnew32*) → *qstore3*  
     *vori*(*qconsnext1, qreadold*) → *qnew32*  
     *State*(*qstore3, qcc3, qnew31, qstore1*) → *qf*  
*TerminalCard*(*qnew26, qstore3, qreadold*) → *qnew31*  
     *vori*(*qvokeynext, qreadcurrent*) → *qvorinext*  
     *State*(*qkvorinext, qcc3, qnew30, qstore1*) → *qf*  
*TerminalCard*(*qconsnext1, qkvorinext, qreadcurrent*) → *qnew30*  
     *State*(*qkvorinext, qcc3, qnew28, qstore1*) → *qf*  
*TerminalCard*(*qnew29, qkvorinext, qreadcurrent*) → *qnew28*  
     *cons*(*qconsnext1, qconsnext1*) → *qnew29*  
     *vori*(*qconsnext1, qreadcurrent*) → *qvorinext*  
     *Key*(*qvorinext*) → *qkvorinext*  
     *State*(*qkvorinext, qcc3, qnew25, qstore1*) → *qf*  
*TerminalCard*(*qnew26, qkvorinext, qreadcurrent*) → *qnew25*  
     *cons*(*qconsnext1, qnew27*) → *qnew26*  
     *cons*(*qconsnext1, qnil*) → *qnew27*  
     *State*(*qstore3, qcc3, qnew38, qstore1*) → *qf*  
     *State*(*qstore3, qcc3, qnew30, qstore1*) → *qf*  
     *State*(*qstore3, qcc3, qnew25, qstore1*) → *qf*  
     *State*(*qstore3, qcc3, qnew28, qstore1*) → *qf*  
     *Hach*(*qkvorinext*) → *qconsnext2*  
     *State*(*qstore3, qcc3, qnew34, qstore1*) → *qf*

---

```

Key(qnew36) → qconsnext2
Key(qnew32) → qconsnext2
Hach(qconsnext2) → qconsnext2
encrypt(qvokeynext, qconsnext2) → qstore3
cons(qvornext, qstore3) → qstore3
read(qnext, qnil) → qreadnext
read(qnext, qreadold) → qreadnext
read(qnext, qreadcurrent) → qreadnext
State(qstore3, qcc3, qnew39, qstore1) → qf
TerminalCard(qstore3, qstore3, qreadnext) → qnew39
vori(qvokeynext, qreadnext) → qvorinext
State(qkvorinext, qcc3, qnew42, qstore1) → qf
TerminalCard(qconsnext1, qkvorinext, qreadnext) → qnew42
State(qkvorinext, qcc3, qnew41, qstore1) → qf
TerminalCard(qnew29, qkvorinext, qreadnext) → qnew41
vori(qconsnext1, qreadnext) → qvorinext
State(qkvorinext, qcc3, qnew40, qstore1) → qf
TerminalCard(qnew26, qkvorinext, qreadnext) → qnew40
State(qstore3, qcc3, qnew41, qstore1) → qf
State(qstore3, qcc3, qnew40, qstore1) → qf
State(qstore3, qcc3, qnew42, qstore1) → qf
read(qnext, qreadnext) → qreadnext

```



D

Automate initial pour *deriv*

## D.1 Automate initial

L'automate suivant reconnaît l'ensemble des conjectures à prouver par  $\mathcal{I}$  à l'aide de *Timbuk* pour l'exemple développé en 7.2.3.

<b>Automaton Initial</b>	
<b>States</b>	
$q[1..12]$ $qe$ $qa$ $qb$ $qc$ $qd$ $qsa$ $qssa$ $qs0$	
$qsc$ $qsd$ $qssd$ $qcp0$ $qcp1$ $qcn0$ $qcn1$ $qf$	
<b>Final States</b>	
$qf$	
<b>Transitions</b>	
$empty$	$\rightarrow qe$
$a$	$\rightarrow qa$
$b$	$\rightarrow qb$
$c$	$\rightarrow qc$
$d$	$\rightarrow qd$
$O$	$\rightarrow q0$
$s(q0)$	$\rightarrow qs0$
$s(qa)$	$\rightarrow qsa$
$s(qsa)$	$\rightarrow qssa$
$s(qc)$	$\rightarrow qsc$
$s(qd)$	$\rightarrow qsd$
$s(qsd)$	$\rightarrow qssd$
$plus(q0, qs0)$	$\rightarrow q1$
$moins(q0, q0)$	$\rightarrow q2$
$dis(q1, q2)$	$\rightarrow q3$
$cn(q3, qe)$	$\rightarrow qcn0$
$plus(q0, qssa)$	$\rightarrow q4$
$moins(q0, qsa)$	$\rightarrow q5$
$dis(q4, q5)$	$\rightarrow q6$
$cn(q6, qcn0)$	$\rightarrow qcn1$
$plus(qb, qs0)$	$\rightarrow q7$
$moins(qb, q0)$	$\rightarrow q8$
$eq(q7, q8)$	$\rightarrow q9$
$cp(q9, qe)$	$\rightarrow qcp0$
$plus(qc, qssd)$	$\rightarrow q10$
$moins(qc, qsd)$	$\rightarrow q11$
$eq(q10, q11)$	$\rightarrow q12$
$cp(q12, qcp0)$	$\rightarrow qcp1$
$deriv(qcn1, qcp1)$	$\rightarrow qf$

# Table des figures

3.1	paires critiques . . . . .	22
3.2	Les règles de déduction permettant de normaliser un problème de filtrage . . . . .	29
3.3	superposition d'arbres . . . . .	30
4.1	schéma de l'algorithme DES . . . . .	49
5.1	Le protocole <i>View Only</i> de <b>SmartRight</b> . . . . .	57
5.2	Spécification du protocole <i>View Only</i> de <b>SmartRight</b> . . . . .	58
5.3	Automate des configurations initiales . . . . .	62
7.1	Système d'inférence $\mathcal{I}$ . . . . .	90



# Index

<b>Symbols</b>	
$\mathcal{Q}$ -instance .....	31
$\mathcal{Q}$ -substitution .....	25
union .....	25
$\mathcal{T}(\mathcal{F})$ -substitution .....	13
$\mathcal{T}(\mathcal{F})/\equiv_{\mathbf{T}}$ .....	77
$\equiv_{\mathbf{T}}$ .....	77
étapes de complétion .....	22
<b>deriv</b> .....	93
codage de .....	94
<b>A</b>	
abstraction .....	22, 35
alphabet .....	12
approximation .....	35
arité .....	12
authentification .....	51
automate d'arbre .....	15
clos .....	24, 26
complétion d'un .....	20, 22
déterministe .....	15
filtrage .....	28
opérations sur les .....	16
taille .....	15
automate d'un terme .....	29
automate des instances .....	31
axiomatisation .....	78–81
<b>B</b>	
Birkhoff	
théorème de .....	77
<b>C</b>	
calcul exact .....	37
clé	
publique .....	48
secrète .....	46
complétion .....	20, 22
réussie .....	24
confidentialité .....	51
configuration .....	15
conséquence inductive .....	76
contexte .....	13
cover .....	88
cryptographie	
asymétrique .....	48
symétrique .....	46
<b>D</b>	
D.E.S .....	46
démonstration .....	74
dérivation .....	89
réussie .....	89
descente infinie .....	86
diséquation .....	77, 86
disponibilité .....	51
<b>E</b>	
ensembles couvrants .....	82
<b>F</b>	
filtrage .....	28
problème de .....	28
solution de .....	28
formule .....	72
atomique .....	72
satisfaction .....	73
<b>H</b>	
Herbrand	
base de .....	75
modèle de .....	76
plus petit modèle de .....	76
structure de .....	75
univers de .....	75
<b>I</b>	
inconsistance .....	79, 80, 82
témoin d' .....	80
intégrité .....	51



## Résumé

L'objectif de cette thèse était d'étendre des techniques de réécriture sur les automates d'arbres afin de pouvoir les utiliser pour étudier des propriétés de sécurité sur les protocoles cryptographiques.

Notre première partie présente la complétion d'automates d'arbres, procédé qui permet de calculer un automate reconnaissant un sur-ensemble des termes atteignables par un système de réécriture à partir d'un langage initial régulier. Nous proposons une condition dite *de linéarité* sur les automates permettant d'étendre le calcul aux systèmes de réécriture non linéaires à gauche, une condition sur la fonction d'approximation permettant de rendre le calcul des descendants exact. D'un point de vue algorithmique, nous présentons un algorithme de recherche des instances d'un terme dans un automate qui accélère la complétion.

Nous avons implémenté ces résultats dans **Timbuk**, qui est un logiciel réalisé en Caml et disponible librement à l'URL <http://www.irisa.fr/lande/genet/timbuk/>.

La deuxième partie de notre travail a été d'utiliser les techniques de complétion décrites précédemment pour vérifier des protocoles cryptographiques. Nous proposons l'exemple de la vérification du protocole *View Only* de **SmartRight** développé par *Thomson Multimédia* et qui vise à protéger des données numériques.

La dernière partie de ce travail s'intéresse à la preuve automatique de propriétés initiales. Nous étudions les méthodes de démonstrations automatiques : récurrence explicite, récurrence par réécriture et preuve par cohérence. Nous proposons en premier lieu un algorithme de preuve de propriétés initiales, avant de montrer comment il est possible d'utiliser la complétion d'automates pour automatiser cet algorithme.

**Mots-clés:** réécriture, preuves, automates, protocoles







# Bibliographie

- [BBC<sup>+</sup>99] B. Barras, S. Boutin, C. Cornes, J. Courant, Y. Coscoy, D. Delahaye, D. de Rauglaudre, J.C. Filliâtre, E. Giménez, H. Herbelin, G. Huet, H. Laulhère, P. Loiseleur, C. Muñoz, C. Murthy, C. Parent, C. Paulin, A. Saïbi, and B. Werner. *The Coq Proof Assistant Reference Manual – Version V6.3*, July 1999.
- [BC03] Yves Bertot and Pierre Casteran, editors. *Le Coq' Art*. 2003. To appear.
- [BKR95] Adel Bouhoula, Emmanuel Kounalis, and Michael Rusinowitch. Automated mathematical induction. *Journal of Logic and Computation*, 5(5) :631–668, 1995.
- [Bol96] An approach to the formal verification of cryptographic protocols. In *3rd ACM Conference on Computer and Communications Security*, pages 106–108, 1996.
- [Bou96] Adel Bouhoula. Using induction and rewriting to verify and complete parameterized specifications. *Theoretical Computer Science*, 170(1–2) :245–276, 1996.
- [Bou97] Adel Bouhoula. Automated theorem proving by test set induction. *Journal of Symbolic Computation*, 23(1) :47–77, 1997.
- [BR95a] Adel Bouhoula and Michael Rusinowitch. Implicit induction in conditional theories. *Journal of Automated Reasoning*, 14(2) :189–235, 1995.
- [BR95b] Adel Bouhoula and Michael Rusinowitch. SPIKE : A system for automatic inductive proofs. In *Algebraic Methodology and Software Technology*, pages 576–577, 1995.
- [CDG<sup>+</sup>97] H. Comon, M. Dauchet, R. Gilleron, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi. Tree automata techniques and applications. Available on : <http://www.grappa.univ-lille3.fr/tata>, 1997.
- [CDGV91] J.L. Coquidé, M. Dauchet, R. Gilleron, and S. Vágvolgyi. Bottom-up tree pushdown automata and rewrite systems. volume 488, pages 287–298, 1991.
- [CJ94] Hubert Comon and Florent Jacquemard. Ground reducibility and automata with disequality constraints. In *Symposium on Theoretical Aspects of Computer Science*, pages 151–162, 1994.
- [CJ97] Hubert Comon and Florent Jacquemard. Ground reducibility is EXPTIME-complete. In *Proc. 12th IEEE Symp. Logic in Computer Science (LICS'97), Warsaw, Poland, June–July 1997*, pages 26–34. IEEE Comp. Soc. Press, 1997.
- [CM96] Evelyne Contejean and Claude Marché. CiME : Completion Modulo *E*. pages 416–419, 1996. System Description available at <http://cime.lri.fr/>.
- [Com91] Hubert Comon. Disunification : A survey. In *Computational Logic - Essays in Honor of Alan Robinson*, pages 322–359, 1991.
- [Com94] H. Comon. Inductionless induction. In René David, editor, *2nd Int. Conf. in Logic For Computer Science : Automated Deduction. Lecture notes*, Chambéry, 1994. Univ. de Savoie.

- [Cor03] Véronique Cortier. *Vérification automatique des protocoles cryptographiques*. PhD thesis, Ecole Normale Supérieure de Cachan, 2003.
- [DJ90] Nachum Dershowitz and Jean-Pierre Jouannaud. Rewrite systems. In *Handbook of Theoretical Computer Science, Volume B : Formal Models and Semantics (B)*, pages 243–320. 1990.
- [DT90] M. Dauchet and S. Tison. The theory of ground rewrite systems is decidable. pages 242–248, June 1990.
- [DY83] D.Dolev and A. Yao. On the security of public key protocols. In *Proc. IEEE Transactions on Information Theory*, pages 198–208, 1983.
- [Gen98] Th. Genet. *Contraintes d'ordre et automates d'arbres pour les preuves de terminaison*. Thèse de Doctorat d'Université, Université Henri Poincaré – Nancy 1, 1998.
- [GK00] T. Genet and F. Klay. Rewriting for Cryptographic Protocol Verification. volume 1831, 2000.
- [GRS01] Valérie Gouranton, Pierre Réty, and Helmut Seidl. Synchronized tree languages revisited and new applications. *Lecture Notes in Computer Science*, 2030 :214–??, 2001.
- [GVTT01] T. Genet and V. Viet Triem Tong. Timbuk Documentation. IRISA / Université de Rennes 1, 2001. <http://www.irisa.fr/lande/genet/timbuk/>.
- [GVTT02] T. Genet and V. Viet Triem Tong. Proving Negative Conjectures on Equational Theories using Induction and Abstract Interpretation. Technical Report RR-4576, 2002.
- [HH82] G. Huet and J.M. Hullot. Proof by induction in equational theories with constructors. *Annual Symposium on Foundations of Computer Sciences*, 1982.
- [Jac96] F. Jacquemard. *Automates d'arbres et réécriture de termes*. PhD thesis, Université Paris-Sud, 1996.
- [JK89] J. Jouannaud and E. Kounalis. Automatic proofs by induction in equational theories without constructors, 1989.
- [KB83] D. E. Knuth and P. B. Bendix. Simple word problems in universal algebras. In J. Siekmann and G. Wrightson, editors, *Automation of Reasoning 2 : Classical Papers on Computational Logic 1967-1970*, pages 342–376. Springer, Berlin, Heidelberg, 1983.
- [KNZ91a] Deepak Kapur, Paliath Narendran, and Hantao Zhang. Automating inductionless induction using test sets. *Journal of Symbolic Computation*, 11(1/2) :81–111, 1991.
- [KNZ91b] Deepak Kapur, Paliath Narendran, and Hantao Zhang. Sufficient-completeness, ground-reducibility and their complexity. In *Acta Informatica*, pages 311 – 350, 1991.
- [KZ95] D. Kapur and H. Zhang. An overview of rewrite rule laboratory (rrl). *J. Computer and Mathematics with Applications*, 29(2) :91–114, 1995.
- [LDG<sup>+</sup>00] X. Leroy, D. Doligez, J. Garrigue, D. Rémy, and J. Vouillon. The Objective Caml system release 3.00 – Documentation and user's manual, 2000. <http://caml.inria.fr/ocaml/htmlman/>.
- [LR97] Sébastien Limet and Pierre Réty. E-unification by means of tree tuple synchronized grammars. *Discrete Mathematics and Theoretical Computer Science*, 1(1) :69–98, 1997.
- [LR98] Sébastien Limet and Pierre Réty. Solving disequations modulo some class of rewrite systems. *Lecture Notes in Computer Science*, 1379 :121–??, 1998.

- 
- [Mea96] Catherine Meadows. The NRL protocol analyzer : An overview. *Journal of Logic Programming*, 26(2) :113–131, 1996.
- [Mus80] David R. Musser. On proving inductive properties of abstract data types. In *Proceedings of the 7th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 154–162. ACM Press, 1980.
- [ORR<sup>+</sup>96] S. Owre, S. Rajan, J.M. Rushby, N. Shankar, and M.K. Srivas. PVS : Combining specification, proof checking, and model checking. In Rajeev Alur and Thomas A. Henzinger, editors, *Computer-Aided Verification, CAV '96*, volume 1102 of *Lecture Notes in Computer Science*, pages 411–414, New Brunswick, NJ, July/August 1996. Springer-Verlag.
- [Pau97] L. Paulson. Proving Properties of Security Protocols by Induction. In *10th Computer Security Foundations Workshop*. IEEE Computer Society Press, 1997.
- [Red90] U. S. Reddy. Term rewriting induction. In M. E. Stickel, editor, *10th International Conference on Automated Deduction*, pages 162–178. Springer, Berlin, Heidelberg, 1990.
- [Rét99] P. Réty. Regular Sets of Descendants for Constructor-based Rewrite Systems. In *Proceedings of the 6th international conference on Logic for Programming and Automated Reasoning (LPAR)*, number 1705 in *Lecture Notes in Artificial Intelligence (LNAI)*, Tbilisi, Republic of Georgia, 1999. Springer Verlag.
- [Sal88] K. Salomaa. Deterministic tree pushdown automata and monadic tree rewriting system. *Journal of Computer and System Sciences*, 1988.