HAL
open science

Extensions des automates d'arbres pour la vérification de systèmes à états infinis

Valérie Murat

To cite this version:

Valérie Murat. Extensions des automates d'arbres pour la vérification de systèmes à états infinis. Performance et fiabilité [cs.PF]. Université de Rennes, 2014. Français. NNT : 2014REN1S033 . tel-01065696

HAL Id: tel-01065696

https://theses.hal.science/tel-01065696v1

Submitted on 18 Sep 2014

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

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

ANNÉE 2014

THÈSE / UNIVERSITÉ DE RENNES 1
sous le sceau de l'Université Européenne de Bretagne

pour le grade de

DOCTEUR DE L'UNIVERSITÉ DE RENNES 1

Mention : INFORMATIQUE

Ecole doctorale MATISSE

présentée par

Valérie MURAT

Préparée à l'unité de recherche UMR 6074 IRISA
Institut de Recherche en Informatique et Systèmes Aléatoires
UFR Informatique Électronique

Extensions d'automates d'arbres pour la vérification de systèmes à états infinis

Thèse soutenue à l'IRISA le 26 juin 2014

devant le jury composé de :

Jean-Michel COUVREUR
Professeur à l'Université d'Orléans
Rapporteur

Pierre-Cyrille HEAM
Professeur à l'Université de Franche Comté
Rapporteur

Thomas JENSEN
Directeur de recherche INRIA à l'IRISA Rennes
Examinateur

Sophie PINCHINAT
Professeur à l'Université de Rennes 1
Examinateur

Yohan BOICHUT
Maître de Conférences à l'Université d'Orléans
Examinateur

Thomas GENET
Maître de Conférences à l'Université de Rennes 1
Directeur de thèse

Remerciements

Par où commencer ? Tout d'abord, je peux raconter que sans Yohan Boichut, ce document n'existerait peut-être pas. Que je n'aurais peut-être pas fait ces 3 années (++) de doctorat. La question étant : dois-je le remercier pour ça ?

Durant l'année de Master 2, il a, par sa grande pédagogie, éveillé en moi la curiosité du travail de recherche, notamment dans le domaine dans lequel se situe cette thèse. Mais la chose pour laquelle il ne m'avait pas préparée, c'est l'inconstance et la difficulté du travail de recherche, le questionnement permanent : est-ce que ce que je fais sert réellement à quelque chose ? En recherche, on n'est jamais vraiment sûr de rien, et c'est quelque chose qu'il faut savoir accepter. Mais j'espère de tout cœur que ce manuscrit sera utile, et dans ce but j'y ai mis tous les efforts pour qu'il soit le plus pédagogique possible. Cette thèse fut une grande aventure, enrichissante, et pour cela Yohan, je te remercie. Et puis tant qu'on y est, je te remercie aussi d'avoir accepté d'être dans mon jury de thèse :-).

Je remercie également Pierre-Cyrille Heam et Jean-Michel Couvreur qui ont accepté d'être les rapporteurs de cette thèse, et dont les commentaires et corrections m'ont permis d'améliorer ce manuscrit. Plus de 220 pages, je compatis... Je remercie aussi Thomas Jensen et Sophie Pinchinat d'avoir accepté de faire partie de mon jury de soutenance, et Tristan Le Gall pour notre collaboration fructueuse.

Pour citer une de mes connaissances, "une thèse sans directeur de thèse, c'est comme un vélo sans guidon". Merci à toi Thomas de m'avoir guidé tout au long de cette thèse, et de m'avoir permis, grâce à ta pédagogie, de comprendre les aspects les plus difficiles du domaine. Je tiens également à remercier l'équipe Celtique pour son accueil durant ces années de thèse, notamment André qui, en plus des échanges intéressants que nous avons eu, m'a rendu de nombreux services.

Cette thèse a été riche par bien des aspects, notamment par les relations que j'ai tissées durant ces 4 ans qui sont devenues de belles relations amicales. Je remercie donc François, pour nos discussions, son enthousiasme permanent, et nos bons moments passés ensemble. Je remercie également Carole et Rouwaida pour nos grandes conversations entre filles, Pierre pour nos sorties concerts, Manu pour les après-midi musicales, Cyrille, Bastien, Lamine... Sans oublier Jean-Christophe pour les discussions et les soirées à Nancy, les potes du LIFO, ainsi que Nicolas, Va, Yves...

Bien entendu je ne peux pas oublier Andrés. Ta présence et ton soutien depuis de nombreuses années me sont nécessaires. Merci pour ton sens de l'humour, ta façon d'être, et ta personnalité qui sortent du commun. Merci également à toi et Nadège de m'avoir hébergé quand j'en ai eu besoin :-p.

Enfin, je remercie Aurore, ma grande bouffée d'oxygène durant cette thèse, qui m'a donné le courage d'affronter ce qu'il fallait affronter, et qui est devenue une grande histoire d'amitié. La miss, tu me manques !

Bien entendu, cette thèse aurait été plus difficile sans une famille présente et chaleureuse. Je remercie alors Alice et Sylvain, Tata Jacqueline et Marie-France, Pépé Jean et Mamé Michou, Mémé Yvette, Tata Madeleine, Tata Marie-Nöelle et Thierry (ouais ouais, je vais citer tout le monde, je fais ce que je veux :-p), Pierre et Audrey, François et Élise, Salomé et Zacharie, Emma, Collyne, Elaïa et le petit dernier, Illan.

Je termine par mes parents, qui ont été d'un grand soutien tout au long de cette aventure et que je ne saurais jamais remercier assez. C'est pour vous que j'ai écrit l'introduction de cette thèse, pour qu'enfin vous compreniez quelque chose à ce que je fais depuis plus de trois ans :-)...

Mathieu, voilà ton tour, et je vais me contenter de citer quelques dates. 6 juillet 2012 : rencontre particulière, enrichissante, étrange... 12 mai 2013 : découverte d'un monde inconnu. Août 2014 : début d'une nouvelle vie. 30 mai 2015 : on se prend pour des stars et on fait une grande fête en notre honneur ;-)... Avec curé et maire causant pour nos beaux yeux :-) ! Merci d'exister, de m'accompagner et me soutenir. Je t'aime.

Enfin, j'aurais voulu pouvoir dire à mon grand-père qui n'est plus là, qu'il est encore aujourd'hui une inspiration. Que parce qu'il rêvait de devenir enseignant, c'était ma grande fierté d'enseigner durant cette thèse.

Alors merci à mon grand-père, naïf invétéré, doux, humaniste, toujours dans la lune –et accessoirement, colonel dans l'armée de terre–, auteur de livres relatant différentes guerres auxquelles il a dû participer.

Car réussir est une guerre. Car ce manuscrit est la preuve d'une victoire.

Musique de Rocky

Valérie

"Vous n'avez cessé d'essayer ?
Vous n'avez cessé d'échouer ?
Alors, essayez encore, échouez encore.
Échouez mieux."

Samuel Beckett

Sommaire

Chapitre 1 : Introduction **7**

Chapitre 2 : Préliminaires **21**
2.1 Mots, termes et systèmes de réécriture 22
2.2 Automates d'arbres 28
2.3 Model-Checking régulier sur arbres 30
2.4 Algorithme de complétion d'automates d'arbres 33
2.5 Timbuk 42
2.6 Treillis et interprétation abstraite 49

Chapitre 3 : État de l'art et comparaisons **55**
3.1 Introduction 56
3.2 Model-Checking régulier et transducteurs 57
3.3 Fonctions d'abstractions du model-checking régulier 69
3.4 Model-Checking avec Maude 79
3.5 Application concrète à l'analyse de programme 92
3.6 Conclusion et ouverture sur les contributions 99

I Automates d'arbres et formules logiques **101**

Chapitre 4 : Caractérisation de points-fixes concluants **103**
4.1 Introduction 105
4.2 Préliminaires 108
4.3 Automates d'arbres symboliques 109
4.4 Solutions de l'algorithme de filtrage pour les STA 114
4.5 À la recherche d'un automate point-fixe concluant 121
4.6 Analyse d'atteignabilité via résolution de formules 131
4.7 Conclusion et perspectives 137

II Automates d'arbres à treillis et interprétation abstraite **139**

Chapitre 5 : Algorithme de complétion pour LTA **141**
5.1 Introduction 143
5.2 Automates d'arbres à treillis (Lattice Tree Automata : *LTA*) 147

5.3 Opérations sur les LTA 152
5.4 Un algorithme de complétion pour les LTA 160
5.5 Complétion pour LTA déroulée sur un exemple 174
5.6 Conclusion 177

Chapitre 6 : Implémentations et expérimentations 179
6.1 Introduction 181
6.2 Préliminaires : fonctionnement de Copster 182
6.3 Implémentation de CopsterLTA et TimbukLTA 195
6.4 Expérimentations 201
6.5 Conclusion et travaux futurs 208

Conclusion et perspectives 211

Bibliographie 215

Introduction 1

Personne ne peut nier la présence, de plus en plus forte, des systèmes informatiques dans la vie de tous les jours : dans les téléphones, les avions, les systèmes médicaux, les protocoles de sécurité sur internet, le vote électronique... Certains de ces systèmes sont critiques et il est donc nécessaire de s'assurer de leur fiabilité. En effet, une erreur pourrait causer des dommages dramatiques aussi bien d'un point de vue financier qu'humain, comme par exemple une défaillance dans le système informatique du pilotage automatique d'un avion. Nous pourrions également citer l'éternel exemple du crash de la fusée Ariane 5 au bout de 37 secondes de vol, dû à une simple erreur du programme informatique.

Prenons plutôt un exemple beaucoup plus récent. Découvert il y a peu par des universitaires finlandais et rendu public par Google, un bug appelé *heartbleed* permet de récupérer des informations confidentielles de nombreux services web connus tels que Yahoo, Flickr, Doodle ou encore Zimbra. Ce bug se situe dans l'*implémentation* de `OpenSSL`, le protocole de sécurité le plus utilisé sur internet, et peut être exploité par n'importe qui. Quand un utilisateur se connecte à un service web de manière sécurisée en utilisant ce protocole, son navigateur web envoie régulièrement des petits messages, appelés *heartbeat*, permettant de vérifier que la connexion est encore fonctionnelle. Ces petits messages contiennent certaines informations, telles que le type du message (ici de type *heartbeat*) ou la taille du message. Le service web confirme la bonne réception de ce message en renvoyant le même contenu au navigateur de l'utilisateur. En donnant une fausse information sur la taille de ce message, la donnée fournie par l'utilisateur est mal interprétée par le service web. Ainsi, dans son message de confirmation, celui-ci va renvoyer à l'utilisateur des informations excédentaires supposées sécurisées et potentiellement sensibles, comme illustré par le site `http://xkcd.com` (voir figure 1.1).

Comment éviter de tels problèmes, sachant que les programmes informatiques sont de plus en plus complexes ? En effet, un programme des plus simples tel qu'un éditeur de texte classique, peut comprendre plus de 1 700 000 de lignes de codes se répartissant sur plus de 400 fichiers. Plusieurs méthodes existent actuellement pour trouver d'éventuelles erreurs. Tout d'abord, nous pouvons citer le *test*, ayant fait ses preuves dans de nombreuses industries. Le test consiste à exécuter un nombre *fini* de fois le programme :

- selon une certaine procédure (appelée *procédure d'exécution*), et
- selon différents paramètres (ensemble de *cas à tester*)

La procédure d'exécution et les cas à tester sont définis au préalable, dans le but de vérifier certaines fonctionnalités du système et d'identifier un nombre maximum de comportements problématiques. Un grand nombre de méthodes ont été définies pour

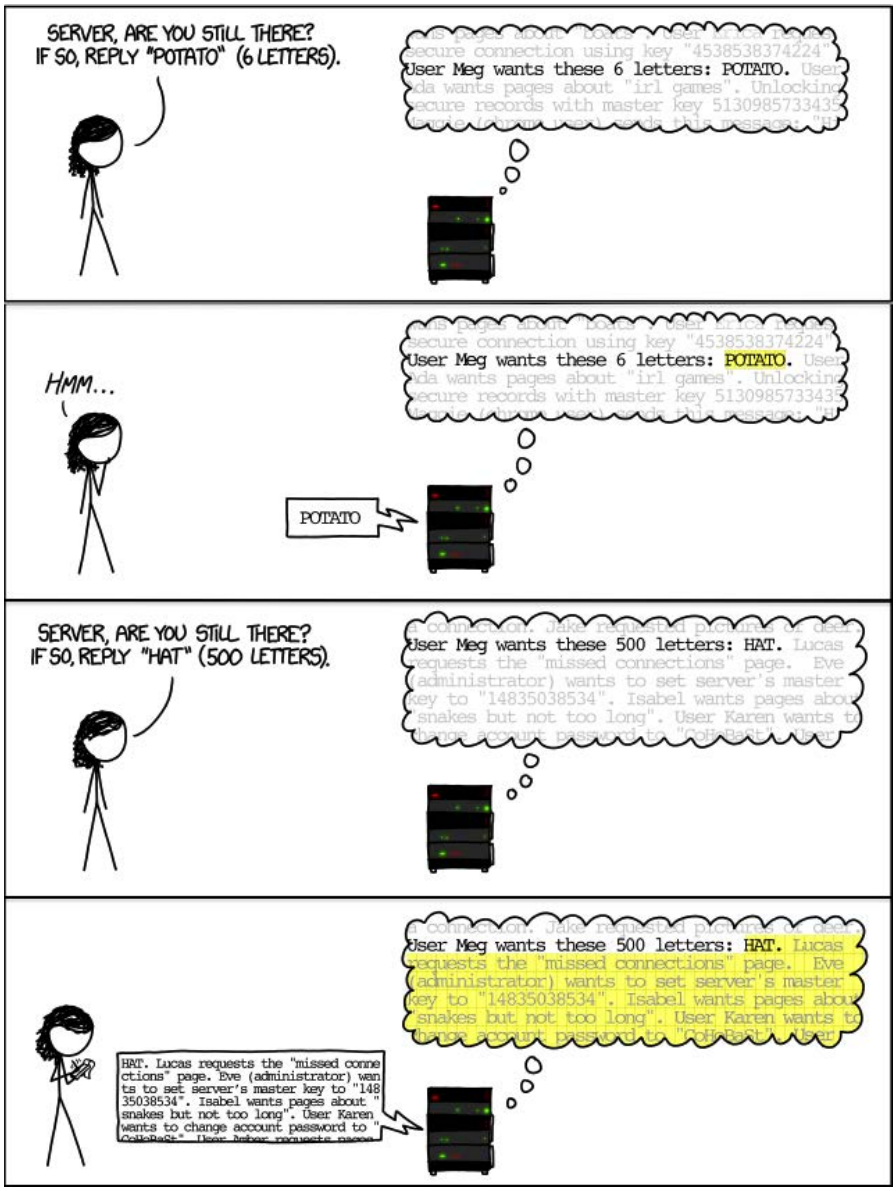


FIGURE 1.1 — Fonctionnement du bug *heartbleed* (source : http://xkcd.com/1354/).

permettre de spécifier au mieux les procédures d'exécution et les différents cas de tests. Cependant, même s'il est relativement simple à mettre en œuvre, le test n'est pas une méthode exhaustive, et un nombre de tests fini ne peut pas couvrir tous les scénarios possibles. En effet, la complexité grandissante des systèmes informatiques rend impossible la prévision de tous les comportements possibles que peut avoir un système. Le test est une méthode non formelle et expérimentale ne garantissant pas la détection de tous les mauvais comportements. Imaginons que l'on teste cent fois le même programme. Alors rien ne permet de garantir l'absence de *bug* : la cent-unième fois comportera peut-être une erreur qui ne sera pas détectée lors des cent premiers tests.

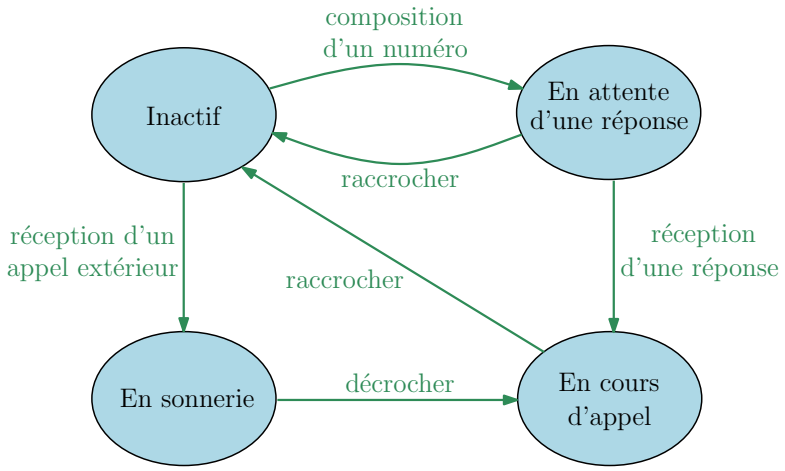


FIGURE 1.2 — Modélisation simplifiée du fonctionnement d'un téléphone sous forme de *système état-transition*.

Vérification formelle

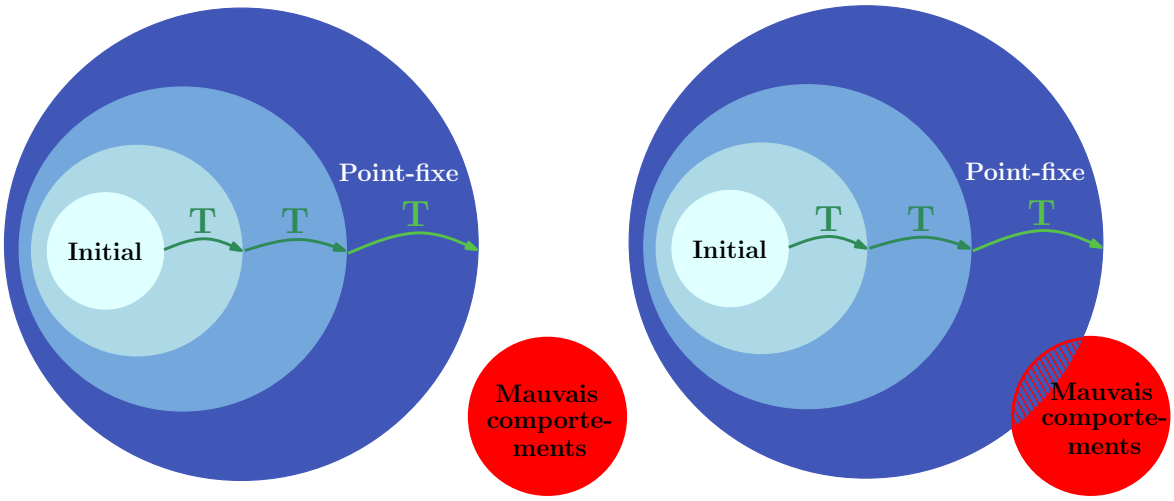
Dans le cas de systèmes critiques, il est donc nécessaire d'ajouter aux tests des méthodes *formelles* fournissant une véritable *preuve* de la fiabilité du système à vérifier. Ces techniques formelles utilisent des méthodes mathématiques permettant de vérifier qu'un système se comporte de manière souhaitée, en calculant *tous* les comportements qu'il peut avoir, et en vérifiant par exemple qu'aucun de ces comportements n'est dangereux.

L'ensemble de ces méthodes définissent des modèles mathématiques représentant fidèlement le système, et parmi ces méthodes, on peut citer :

- La *preuve de programme* : on démontre, *manuellement* ou grâce à un outil d'aide à la preuve automatique (tel que Coq [Bertot et Castéran, 2004] ou Isabelle [Paulson, 2008]), que le système considéré se comporte comme prévu. Cette approche consiste à poser le problème de vérification comme la preuve d'un théorème, en utilisant des règles de déduction. Elle permet la vérification de nombreux systèmes, mais nécessite une intervention souvent fastidieuse du vérificateur : il est donc souvent indispensable que ce vérificateur soit un *expert*.
- Le *Model-Checking* consiste à vérifier *automatiquement* chaque exécution, chaque comportement possible du système. Pour cela, le Model-Checking utilise une représentation sous forme de *système état-transition*. Les *états* représentent les différents comportements possibles du système, et les transitions représentent les actions permettant de passer d'un état à un autre. La figure 1.2 représente une modélisation simplifiée (sans répondeur) d'un téléphone, sous forme de système état-transition. Un téléphone peut être inactif, en train de sonner, en attente de réponse (après avoir composé un numéro), ou en cours d'appel. Ces différents comportements constituent donc les *états* du système.

Propriétés à vérifier

Dans le cadre des méthodes formelles, la vérification du bon comportement d'un système est effectué en s'assurant qu'il respecte certains critères formalisés sous forme de *propriétés*. Le Model-Checking, qui constitue principalement le cadre de cette thèse,



(a) La propriété est vérifiée (b) La propriété n'est pas vérifiée

FIGURE 1.3 — Calcul de l'ensemble des états accessibles afin de vérifier une propriété de *sûreté*.

permet d'établir un diagnostic précis pour une propriété donnée et retourne un contre-exemple quand elle n'est pas vérifiée. Il existe plusieurs types de propriétés, et les principales sont les suivantes :

- Les propriétés de *sûreté* consistent à vérifier qu'un mauvais comportement n'arrivera jamais. Prenons par exemple une imprimante en réseau, utilisée par plusieurs personnes. Une propriété de sûreté consisterait par exemple à vérifier qu'il n'y a pas de mélange entre plusieurs impressions différentes, *i.e.* que les feuilles d'une impression x ne sont pas imprimées en plein milieu d'une impression y.
- Les propriétés de *vivacité* consistent à vérifier qu'une situation va forcément se produire un jour. Reprenons l'imprimante en réseau : si un utilisateur a lancé une impression, alors cette impression va forcément s'effectuer un jour.

Cette thèse considère essentiellement des *propriétés de sûreté*. Pour vérifier une propriété de sûreté dans le cadre du Model-Checking, il faut calculer l'ensemble des comportements que le système peut atteindre (ou encore l'ensemble des *états* dans lequel le système peut se trouver), et vérifier qu'aucun de ces comportements n'est mauvais, *i.e.* ne respecte pas la propriété. Dans le cas très simple du téléphone (figure 1.2), tous les comportements ou états du système sont connus. Mais dans la majorité des cas, la complexité des systèmes informatiques implique l'impossibilité de prévoir tous les comportements possibles d'un système : on connait alors seulement leurs états *initiaux*. Grâce à une *fonction de transition* modélisant toutes les *actions* que le système peut effectuer, il s'agit ensuite de calculer, en passant par des étapes de calcul intermédiaires, l'ensemble de tous les *états accessibles*, *i.e.* que le système est susceptible d'atteindre un jour. La figure 1.3 représente ce calcul des états accessibles d'un système :

- **Initial** représente l'ensemble des *états initiaux* du système.
- $\xrightarrow{T}$ représente la *fonction de transition* calculant, à partir des *états initiaux* et des actions que le système peut effectuer, tous les comportements que le système pourra avoir à l'étape d'après, *i.e.* les états directement accessibles.
- Ce calcul est itéré jusqu'à ce qu'il ne soit plus possible d'ajouter un nouvel état. On arrive alors à ce que l'on appelle un **Point-fixe**.

Une fois ce *point-fixe*, contenant tous les états possibles du systèmes, calculé, la vérification d'une propriété de sûreté consiste alors à vérifier qu'aucun mauvais état ou mauvais comportement n'en fait partie. Sur la figure 1.3a, la propriété est vérifiée puisque l'ensemble des états accessibles n'admet aucun mauvais comportement. En revanche, la figure 1.3b présente une violation de la propriété par le système. Il s'agit alors d'un problème d'*atteignabilité*. Dans le cas ou ce nombre d'états est fini, le calcul de tous les états accessibles termine, et le problème d'atteignabilité est alors *décidable*.

Systèmes à nombre d'états *infini*

La vérification de systèmes intéressants –tels que les protocoles de sécurité ou encore les programmes $Java$– nécessite de considérer un nombre d'états *infini*. Par exemple, si le système vérifié travaille sur des entiers, il est plus générique de considérer un ensemble *infini* d'entiers. En effet, chaque ordinateur possédant une borne sur les entiers, on pourrait alors imaginer ne considérer qu'un ensemble fini d'entiers pour le problème de vérification. Cependant, cette borne est différente selon le matériel utilisé et l'usage d'un ensemble *infini* d'entiers permet d'avoir une méthode de vérification générique ne dépendant pas de la machine sur laquelle le système est exécuté.

Contrairement au cas d'un espace d'états fini, le problème de vérification dans le cas infini est bien souvent *indécidable*. Prenons par exemple un échange de messages entre un utilisateur, que l'on appellera Alice, et un serveur mail ou un site marchand que l'on appellera Bob. Ces messages contiennent parfois des données sensibles telles qu'un mot de passe ou un numéro de carte de bleue comme nous le voyons en figure 1.4 : l'échange doit donc être sécurisé. Dans ce but, les messages sont chiffrés par des clés, et les clés de déchiffrement sont connues uniquement d'Alice et de Bob afin d'assurer la confidentialité des données.

Dans de tels échanges de messages, il est intéressant de vérifier qu'il est impossible pour une tierce personne de récupérer une information confidentielle ou une clé de déchiffrement. Une tierce personne (un intrus malveillant) peut tenter de récupérer des données secrètes en envoyant, sous une *fausse identité*, des messages à Alice ou à Bob (voir figure 1.4). Si les réponses reçues par l'intrus lui permettent d'accéder à des informations confidentielles (comme c'est le cas dans le bug *heartbleed*), alors le protocole possède une faille qu'il est souhaitable de trouver en vérifiant le système.

Pour vérifier que le protocole ne possède aucune faille, *i.e.* qu'aucun mauvais comportement tel que la récupération de données secrètes par l'intrus n'est accessible, il faudrait pouvoir calculer tous les messages possibles que l'intrus peut fabriquer. Mais l'intrus, même s'il possède peu d'informations au début, peut éventuellement enrichir sa connaissance grâce aux réponses d'Alice ou de Bob, si elles contiennent des données intéressantes que l'intrus peut utiliser (voir figure 1.4). Ainsi, il est possible pour l'intrus de construire une infinité de messages différents, et pour vérifier qu'aucun de ces messages n'est dangereux, il faudrait alors être capable de tous les calculer, ce qui est bien souvent un problème *indécidable*.

La vérification des protocoles de sécurité ainsi que de la plupart des systèmes à états infinis pose donc deux problèmes difficiles :

(1) Comment manipuler et représenter des ensembles infinis ?

(2) Comment calculer des ensembles infinis en un temps fini ?

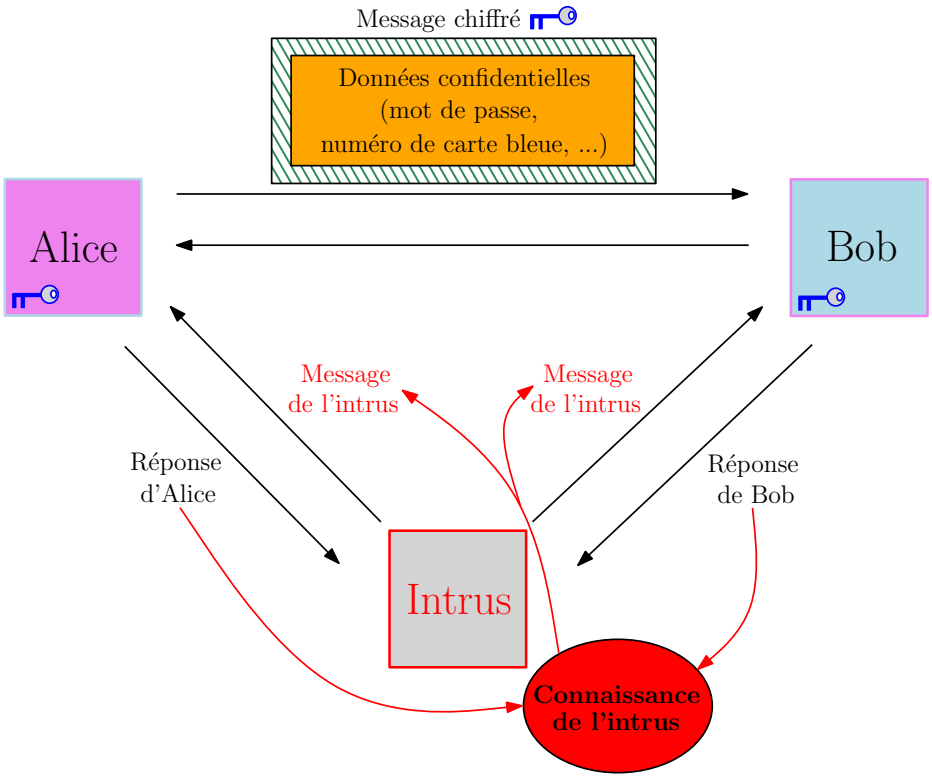


FIGURE 1.4 — Un intrus malveillant peut construire une *infinité* de messages différents pour tenter de récupérer des données confidentielles.

Une représentation *régulière* pour les ensembles *infinis*

Pour résoudre le problème (1), récurrent en vérification, il n'existe que des solutions partielles. Une de ces solutions consiste à représenter chaque état du système par une séquence possiblement *infinie* de *symboles* : cette séquence est alors appelée un *mot*. L'ensemble des symboles permettant de former un mot est défini au préalable, et constitue ce que l'on appelle l'*alphabet*. Soit $\Sigma = \{0, 1\}$ un alphabet contenant le symbole 0 et le symbole 1 : on peut alors construire les mots 10, 111, 1001010, *etc.*, soit l'ensemble *infini* des nombres *binaires*.

Dans ce contexte, les ensembles *infinis* d'états, qui sont alors des ensembles infinis de *mots*, sont représentés par une structure finie et symbolique. Cette structure *générique* est appelée *automate fini*. Cependant, le problème (1) étant indécidable, les *automates finis* ne permettent de représenter qu'une classe limitée d'ensembles infinis : les ensembles *réguliers*. Malgré cette contrainte, cette représentation en automate est généralement suffisante pour représenter les ensembles d'états accessibles de nombreux systèmes.

La figure 1.5 représente un automate fini construit sur l'alphabet $\Sigma = \{0, 1\}$. Cet automate reconnait l'ensemble *infini* des nombres *pairs* en codage binaire. En effet, un nombre binaire est *pair* s'il termine par un 0. Or cet automate permet de reconnaitre les mots comprenant :

– N'importe quelle séquence de symboles 0 ou 1. Ceci est représenté par la boucle $\overset{0,1}{\circlearrowright}$: il peut y avoir de zéro jusqu'à une infinité de 0 ou de 1.

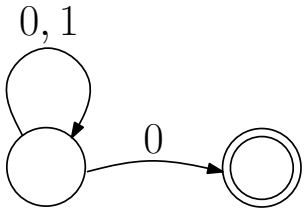


FIGURE 1.5 — Automate représentant l'ensemble des nombres binaires *pairs*.

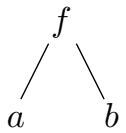
– Puis le mot reconnu termine forcément par le symbole 0 (transition $\xrightarrow{0}$). Les nombres ainsi représentés sont donc forcément pairs.

L'ensemble de mots (ou états) représentés est alors *infini* et comprend par exemple les mots 10, 110, 1010, 111101010, *etc*. Un ensemble de mots reconnus par un automate $\mathcal{A}$ est nommé le *langage* de $\mathcal{A}$.

Le calcul des états accessibles utilisant les *langages réguliers* pour représenter des ensembles *infinis* d'états, est appelé Regular Model-Checking (Model-Checking *régulier*) [Bouajjani et al., 2000].

Termes et systèmes de réécriture

La représentation d'un état par un mot peut avoir des limites. Il existe alors une autre modélisation se construisant à partir de symboles qui, contrairement au cas des mots, possèdent une *arité*. Ces symboles, dit fonctionnels, permettent de construire des structures en *arbre* : en effet, l'arité de chaque symbole fonctionnel signifie le nombre de *fils* que ce symbole devra posséder. Prenons par exemple le symbole f d'arité 2 : f doit alors posséder deux fils. Rajoutons les symboles a et b d'arité 0 que l'on appelle alors *constantes*. À partir de ces trois symboles on peut représenter l'arbre suivant :



Cet arbre peut également s'écrire sous la forme d'une expression $f(a, b)$ et constitue alors ce que l'on appelle un *terme*. La modélisation en arbre est très utilisée en informatique théorique, pour représenter, par exemple, la structure des phrases pour le traitement automatique des langues, mais aussi pour représenter la structure d'un document `XML`, les états d'un programme $Java$, ou encore le fonctionnement d'un protocole de sécurité. En effet, les messages envoyés durant le protocole sont facilement représentables par des termes de la forme *message(Alice, Bob, donnée envoyée)* pour un message envoyé d'Alice à Bob. La donnée envoyée peut également se représenter sous forme de terme : si cette donnée est confidentielle, elle sera chiffrée par une clé (voir figure 1.4), et représentée par le terme *chiffrement(clé(Bob), donnée secrète)*, si la clé utilisée pour protéger le message est celle de Bob. La figure 1.6 présente la modélisation, sous forme d'arbre, d'un message envoyé entre Alice et Bob contenant une une donnée secrète chiffrée par la clé de Bob.

La modélisation choisie dans cette thèse représente les états d'un système par des *termes*, et les ensembles –éventuellement infinis– d'états (donc de termes) par des *automates d'arbres*. Les automates d'arbres reconnaissent des *langages réguliers de termes*.

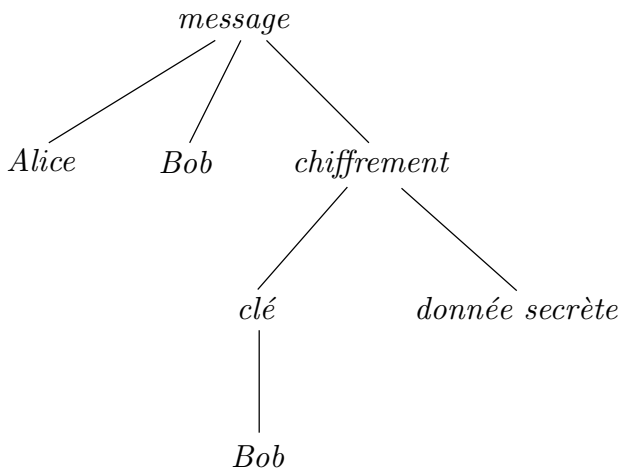


FIGURE 1.6 — Représentation en arbre d'un message envoyé entre Alice et Bob.

Le calcul de l'ensemble des états accessibles dans le cas où les ensembles *infinis* d'états sont représentés par des *automates d'arbres*, est appelé Tree Regular Model-Checking (Model-Checking régulier sur *arbres*) [Abdulla et al., 2002].

Rappelons que pour calculer l'ensemble des états accessibles d'un système, il faut appliquer une fonction de transition sur l'ensemble des états initiaux (figure 1.3). Dans le cas où les ensembles d'états sont des *automates d'arbres*, cette fonction de transition peut être un *transducteur d'arbres* [Abdulla et al., 2002], ou un *système de réécriture*, choix que nous avons retenu dans cette thèse. Un système de réécriture est un ensemble de règles de réécriture $l \to r$ où l et r sont des termes *non-clos* : autrement dit, ils peuvent contenir des variables. On peut par exemple avoir le système de réécriture suivant : $\mathcal{R} = \{f(x,y) \to f(g(x), g(y))\}$, ne comportant qu'une seule règle de réécriture, où x et y sont des variables, et f et g sont des symboles de l'alphabet. Soit $l \to r$ une règle de réécriture. Alors cette règle s'applique à un terme t si celui-ci contient une *instance* des variables du membre gauche l de la règle. On dit alors que t se réécrit en un terme t', où t' est obtenu en remplaçant les variables du membre droit r par l'instance trouvée pour l. Prenons la règle de réécriture $f(x,y) \to f(g(x), g(y))$ et le terme $f(a,b)$ vu plus haut. Alors une instance possible est $x \mapsto a$ et $y \mapsto b$. Ainsi, le terme $f(a,b)$ est réécrit en $f(g(a), g(b))$. On écrit alors :

$$f(a,b) \to_{\mathcal{R}} f(g(a), g(b)).$$

De la même manière, le terme $f(g(a), g(b))$, avec l'instance $x \mapsto g(a)$ et $y \mapsto g(b)$, peut se réécrire en $f(g(g(a)), g(g(b)))$, et ainsi de suite.

L'application d'une règle de réécriture sur *un seul* terme est assez simple, comme nous venons de le voir. Cependant, le calcul des états accessibles d'un système nécessite que le système de réécriture soit appliqué sur un ensemble potentiellement *infini* de termes, *i.e.* un *automate d'arbres*, tel que présenté en figure 1.7.

L'application d'un système de réécriture sur un automate d'arbre nécessite un algorithme appelé ***complétion d'automates d'arbres***, qui a tout d'abord été défini dans [Genet, 1998], puis amélioré dans [Feuillade et al., 2004] et dans [Genet et Rusu, 2010]. Nous allons spécifier et étendre cet algorithme tout au long de cette thèse. Le langage de l'automate *point-fixe* calculé grâce à cet algorithme ($\mathcal{A}_{pf}$ en figure 1.7) contient

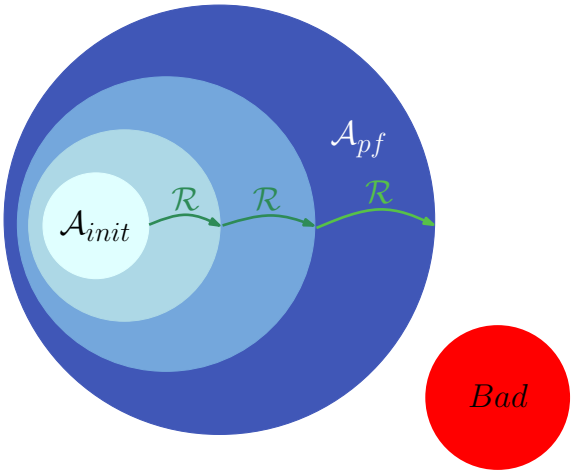


FIGURE 1.7 — Calcul des accessibles par ***complétion d'automates d'arbres***, depuis un automate d'arbres initial $\mathcal{A}_{init}$ et un système de réécriture $\mathcal{R}$.

l'ensemble des termes atteignables par le système. Notons que dans ce cas, l'ensemble des mauvais comportements est alors un ensemble de *termes interdits* que nous notons Bad (voir figure 1.7). Cet ensemble de termes interdits peut être représenté par un automate d'arbres, notamment si cet ensemble est infini. Ainsi, une propriété de sûreté est vérifiée si l'automate $\mathcal{A}_{pf}$ ne reconnait aucun terme de Bad, ou autrement dit, si l'*intersection* entre le langage de l'automate point-fixe et l'ensemble Bad est *vide*. Plus formellement, si $\mathcal{L}(\mathcal{A}_{pf})$ dénote le langage de $\mathcal{A}_{pf}$, alors une propriété de sûreté est vérifiée si $\mathcal{L}(\mathcal{A}_{pf}) \cap Bad = \emptyset$.

Abstraction du calcul de point-fixe : calculer en un temps fini un ensemble infini

Résumons la modélisation choisie dans cette thèse pour répondre partiellement au problème difficile (1) : nous représentons un ensemble infini d'états par un *automate d'arbres*, se restreignant alors à des langages *réguliers* de termes, et la fonction de transition choisie est un *système de réécriture*. Qu'en est-il du problème indécidable (2) ? En effet, l'ensemble des états accessibles d'un système n'est pas toujours calculable et l'algorithme de ***complétion d'automates d'arbres*** peut ne jamais terminer.

Reprenons l'exemple de la section précédente, où $\mathcal{R} = \{f(x,y) \to f(g(x),g(y))\}$ est un système de réécriture, et $f(a,b)$ un terme. Alors nous avons vu que $f(a,b) \to_{\mathcal{R}} f(g(a),g(b)) \to_{\mathcal{R}} f(g(g(a)),g(g(b))$. La règle de réécriture peut encore s'appliquer sur ce dernier terme, puis sur le suivant, et ainsi de suite, et on a alors la chaine infinie de réécriture :

$$f(g(g(a)),g(g(b)) \to_{\mathcal{R}} f(g(g(g(a))),g(g(g(b)))) \to_{\mathcal{R}} f(g(g(g(g(a)))),g(g(g(g(b))))) \to_{\mathcal{R}} \dots$$

Aussi la réécriture à partir du terme initial $f(a,b)$ ne termine jamais. De manière intuitive, on peut en déduire que de la même façon, l'application d'un système de réécriture à partir d'un automate d'arbres initial $\mathcal{A}_{init}$ (effectuée grâce à l'algorithme de *complétion*) peut ne pas converger : le point-fixe $\mathcal{A}_{pf}$ n'est alors jamais calculé et la vérification d'une propriété de sûreté se révèle impossible dans ce cas. Une solution

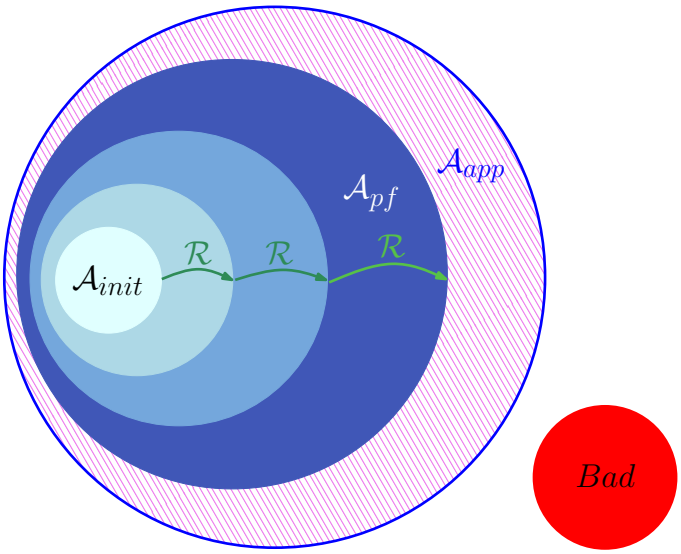


FIGURE 1.8 — Abstraction du calcul de point-fixe par complétion d'automates d'arbres : calcul d'une *sur-approximation*.

à ce problème consiste à *abstraire* le calcul de complétion, afin de calculer une approximation de l'ensemble des termes accessibles, soit une approximation du point-fixe calculable en un temps *fini*. Rappelons que la vérification d'une propriété de sûreté, dans la représentation choisie, consiste à vérifier qu'il n'y a aucun terme interdit dans l'ensemble des termes accessibles. Il faut donc que cette approximation contienne l'ensemble des termes accessibles : il s'agit alors d'une *sur-approximation*. Si la propriété est vérifiée pour un ensemble plus grand de termes, alors elle l'est forcément pour le système vérifié.

C'est ce que nous pouvons voir en figure 1.8 : quand le calcul de l'automate point-fixe $\mathcal{A}_{pf}$ ne converge pas au bout d'un certain temps, une sur-approximation, représentée par l'automate d'arbres $\mathcal{A}_{app}$, est calculée et contient l'ensemble des termes accessibles. On a alors $\mathcal{L}(\mathcal{A}_{pf}) \subseteq \mathcal{L}(\mathcal{A}_{app})$. Grâce à cette inclusion, on peut déduire de manière triviale que si l'intersection entre la sur-approximation et l'ensemble de termes interdits est vide, alors elle l'est également entre le point-fixe et l'ensemble des termes interdits. Autrement dit, $\mathcal{L}(\mathcal{A}_{app}) \cap Bad = \emptyset \implies \mathcal{L}(\mathcal{A}_{pf}) \cap Bad = \emptyset$.

Le calcul d'une *sur-approximation* des états accessibles grâce à une fonction d'abstraction, dans le cas où les ensembles d'états infinis sont représentés par des automates d'arbres, est appelé Abstract Tree Regular Model-Checking (Model-Checking abstrait régulier sur arbres) [Bouajjani et al., 2006a].

Ce calcul d'une sur-approximation est effectué grâce à une *fonction d'abstraction*, qui va permettre l'*accélération* et la *convergence* du calcul. Cette fonction d'abstraction est appliquée à *chaque étape* du calcul. Il existe plusieurs fonctions d'abstraction que nous détaillerons dans cette thèse, et ces fonctions sont généralement définies *manuellement* par le vérificateur. La fonction d'abstraction retenue dans cette thèse est un ensemble d'équations définies sur des termes *non-clos* (contenant des variables). Par exemple, $E = \{g(x) = g(g(x))\}$ est un ensemble possédant une seule équation. Ces équations permettent de générer des classes d'équivalence entre les différents *états* d'un automate d'arbres, puis les états équivalents sont alors *fusionnés* : la diminution du nombre d'états force alors la convergence du calcul. Donnons-en l'intuition sur les termes. Reprenons

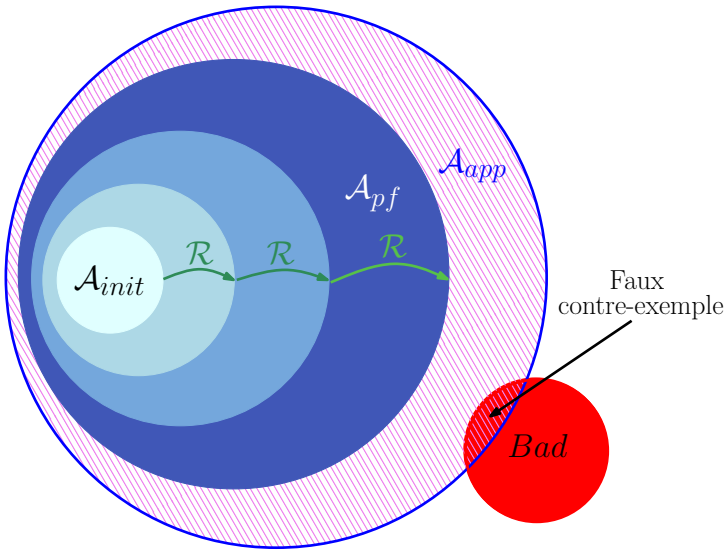


FIGURE 1.9 — Faux contre-exemple dû à une sur-approximation trop grossière.

le système de réécriture $\mathcal{R} = \{f(x,y) \to f(g(x), g(y))\}$. Nous avons vu que cette règle de réécriture génère une infinité de termes :

$$f(a,b) \to_{\mathcal{R}} f(g(a), g(b)) \to_{\mathcal{R}} f(g(g(a)), g(g(b)) \to_{\mathcal{R}} f(g(g(g(a))), g(g(g(b)))) \to_{\mathcal{R}} \dots$$

L'équation $E : g(x) = g(g(x))$ permet de créer deux *classes d'équivalence* :

- La première contient tous les termes $g(a), g(g(a)), g(g(g(a)))$, *etc.* : on peut alors la noter $[g(a)]_E$.
- La seconde contient tous les termes $g(b), g(g(b)), g(g(g(b)))$, *etc.* : on peut alors la noter $[g(b)]_E$.

Ainsi, tous les termes calculés par réécriture sont déjà compris dans les classes d'équivalence. La chaine de réécriture s'abstrait et termine donc de la manière suivante :

$$f(a,b) \to_{\mathcal{R}} f([g(a)]_E, [g(b)]_E) \xrightarrow{\mathcal{R}}$$

L'algorithme de complétion d'automate d'arbres, enrichi de fonctions d'abstractions, a été implémenté dans un outil appelé Timbuk [Genet et Viet Triem Tong, 2001b, Genet, 2008], et a déjà permis la vérification de protocoles de sécurité ou de programmes $Java$.

Présentation des contributions et structure de la thèse

Un premier problème de l'algorithme de complétion d'automate d'arbres est la finesse de l'abstraction : en effet, si la sur-approximation est trop grossière, elle peut reconnaitre des termes interdits qui ne font pas partie de l'ensemble des termes accessibles par le système, comme nous pouvons le voir en figure 1.9. Les termes interdits ainsi reconnus sont alors des *faux contre-exemples*. Autrement dit, la sur-approximation ne vérifie pas la propriété de sûreté, alors que le système la vérifie.

Dans ce cas, on peut faire appel à des techniques de *raffinement* de l'abstraction : l'abstraction est affinée afin de ne plus accepter de faux contre-exemple. Bien que ces

procédures soient automatiques, elles sont coûteuses, et ne garantissent pas l'obtention d'une approximation suffisamment précise pour répondre au problème de vérification. La définition de fonctions d'abstractions intéressantes, permettant à la fois de faire converger le calcul et de ne pas admettre de faux contre-exemple, est un problème difficile, de par le paramétrage *manuel* de l'abstraction. En effet, ces fonctions sont données par l'utilisateur et nécessitent donc une haute expertise de celui-ci au niveau de l'algorithme de complétion autant que dans le type de système vérifié.

Pour répondre à ce problème, notre première contribution permet de caractériser, par des formules logiques et de manière *automatique*, ce qu'est une "bonne" sur-approximation : une approximation représentant un sur-ensemble des configurations accessibles, et qui soit suffisamment précise pour ne pas reconnaitre de faux contre-exemples. À partir d'un automate d'arbres initial $\mathcal{A}_0$, d'un ensemble de termes interdits Bad et un système de réécriture $\mathcal{R}$, les formules logiques caractérisant un point-fixe n'admettant pas de termes de l'ensemble Bad sont générées grâce à un nouveau type d'automate appelé *automate d'arbres symbolique*. Résoudre ces formules, *pour une taille d'automate d'arbres symbolique donnée*, permet alors de trouver une sur-approximation concluante de *taille équivalente*, sans avoir recours à aucun paramétrage manuel. Si aucune solution n'est trouvée pour la taille donnée, alors la taille de l'automate d'arbres symbolique est augmentée et le calcul des formules est répété. Cette contribution fait l'objet de la partie I contenant le chapitre 4.

Le second problème est le passage à l'échelle, autrement dit le temps de calcul parfois élevé du calcul de complétion quand on s'attaque à des problèmes de la vie courante. Dans les travaux sur la vérification de programmes $Java$ utilisant la complétion [Boichut et al., 2006], l'explosion du calcul de complétion peut notamment être due au traitement des entiers. En effet, dans ces travaux, les entiers sont représentés sous forme de termes par des *entiers de Peano*. Un entier de Peano est constitué à partir de trois symboles : $succ$, pour dénoter le successeur d'un nombre, $pred$ pour son prédécesseur, et $zero$ pour l'entier 0. L'entier 2 est alors représenté par le terme $succ(succ(zero))$, et l'entier -1 par le terme $pred(zero)$. Les opérations arithmétiques sur de tels entiers sont effectuées par des règles de réécriture : l'exécution d'une opération arithmétique peut alors nécessiter l'application de centaines de règles de réécriture. Par exemple, l'opération $300 + 250$ nécessite 300 applications de règles de réécriture.

L'idée de notre seconde contribution est alors de pouvoir se passer de centaines d'applications de règles de réécriture en évaluant directement le résultat d'une opération arithmétique. Dans ce cas, les entiers sont représentés tels quels, sans passer par une représentation en entier de Peano. D'une façon plus générale, il s'agit d'intégrer des éléments d'un domaine *infini*, tel que l'ensemble $\mathbb{Z}$, dans un automate d'arbres. Cependant, cette représentation pose un autre problème. Le problème de convergence lié à la structure d'un terme peut être capturé grâce au techniques d'abstraction existantes (telles qu'un ensemble d'équation) comme nous l'avons déjà expliqué, mais comment traiter le possible problème de convergence lié à l'intégration d'un domaine *infini* dans l'algorithme de complétion ?

Pour cela, nous utilisons des méthodes issues du domaine de l'*interprétation abstraite*. L'interprétation abstraite est une théorie introduite par [Cousot et Cousot, 1977] permettant de trouver une approximation de la sémantique d'un programme. L'abstraction du programme se fait généralement par une abstraction des domaines utilisés par

le programme à l'aide d'un *treillis*, *i.e.* un ensemble partiellement ordonné. Un exemple de treillis connu de tous est l'ensemble des intervalles définis sur les entiers.

Dans la partie II de cette thèse, nous adaptons à notre problème l'idée décrite dans [Le Gall et Jeannet, 2007], en intégrant les éléments d'un treillis abstrait dans un automate d'arbres, appelé alors *automate d'arbres à treillis* ou LTA (*Lattice Tree Automata*). Dans cette partie, nous étendons également les opérations standards des automates d'arbres classiques aux LTA telles que l'union, l'intersection et la déterminisation. Enfin, en tant que contribution principale, nous définissons un nouvel algorithme de complétion, équipé d'une fonction d'abstraction, adapté à ce nouveau type d'automates d'arbres. Nous verrons également que l'intérêt principal de cet algorithme est qu'il ne dépend pas du treillis utilisé : cet algorithme est *générique* et également *paramétrable*. Ainsi, n'importe quel treillis (respectant certaines conditions qui seront explicitées), peut être *branché* de façon transparente dans cet algorithme afin d'abstraire différents types de domaines infinis.

Le chapitre 5 décrit le contexte formel et algorithmique de cette méthode, tandis que le chapitre 6 décrit l'implémentation de notre contribution dans un outil appelé TimbukLTA, adapté de Timbuk, ainsi que son application à la vérification de programmes $Java$.

Avant de décrire nos contributions, le chapitre 2 des Préliminaires permet de décrire le contexte formel de cette thèse, et le chapitre 3 permet de situer la technique de complétion avec abstraction par rapport à d'autres techniques similaires.

Publications

WORKSHOP INTERNATIONAL AVEC COMITÉ DE LECTURE
(1) Y. Boichut, T.-B.-H. Dao et V. Murat. **Characterizing Conclusive Approximations by Logical Formulae**. *Reachability Problems (RP'11), LNCS 6945. 2011.*

CONFÉRENCE INTERNATIONALE AVEC COMITÉ DE LECTURE
(2) T. Genet, T. Le Gall, A. Legay et V. Murat. **A Completion Algorithm for Lattice Tree Automata**. *Proceedings of 18th International Conference on Implementation and Application of Automata (CIAA'13), LNCS 7982. 2013.*

RAPPORT TECHNIQUE
(3) T. Genet, T. Le Gall, A. Legay et V. Murat. **Tree Regular Model Checking of Lattice-Based Automata**. *Technical Report RT-0424, INRIA. 2012.*

Le chapitre 4 a fait l'objet de la publication (1), et les chapitres 5 et 6 ont fait l'objet des publications (2) et (3).

Préliminaires

2

Sommaire

2.1 Mots, termes et systèmes de réécriture **22**

2.2 Automates d'arbres **28**

2.3 Model-Checking régulier sur arbres **30**

2.4 Algorithme de complétion d'automates d'arbres **33**

2.4.1 Une étape de complétion 34

2.4.2 Abstraction par équations 38

2.4.3 Algorithme de complétion avec abstraction par équation . . . 41

2.5 Timbuk **42**

2.6 Treillis et interprétation abstraite **49**

2.6.1 Treillis et leurs propriétés 49

2.6.2 Connexions de Gallois 52

2.6.3 Opérateur de *widening* 53

"Je suis jeune il est vrai, mais aux âmes bien nées,
La valeur n'attend point le nombre des années."

Pierre Corneille, *Le Cid*

Ce chapitre permet de rappeler les bases et définitions nécessaires à la compréhension de cette thèse. Tout d'abord un rappel sur les mots, termes, et systèmes de réécriture, ainsi que sur les équations sur termes utilisées dans les méthodes d'approximations, puis sur les automates d'arbres. Ces définitions proviennent principalement des travaux [Baader et Nipkow, 1998, Comon et al., 2007, Genet, 2009].

Nous enchainerons ensuite sur le principe du Model-Checking régulier sur arbres, utilisant automates d'arbres et systèmes de réécriture, puis nous détaillerons l'algorithme de complétion au coeur de cette technique. En effet, cette thèse consistera en partie à modifier ou utiliser cet algorithme de complétion.

Nous finirons ensuite par quelques notions et rappels sur l'interprétation abstraite [Cousot et Cousot, 1977], qui seront utilisés principalement à partir du chapitre 5.

2.1 Mots, termes et systèmes de réécriture

Mots, automates de mots

Définissons tout d'abord la base de la modélisation par automate : les mots, les automates de mots et le langage reconnu par un automate.

DÉFINITION 2.1.1 (Alphabet et mot)
Un alphabet Σ est un ensemble fini de symboles. Un mot est une succession finie de symboles de Σ.

EXEMPLE *2.1*
Soit $\Sigma = \{a, b\}$. Alors $abbaabab$ est un mot construit sur cet alphabet. ◀

DÉFINITION 2.1.2 (Automate fini)
Un automate fini de mot est un tuple $\mathcal{A} = \langle \Sigma, \mathcal{Q}, \mathcal{Q}_0, \mathcal{Q}_f, \delta \rangle$ avec $\mathcal{Q}$ un ensemble fini d'états, Σ un alphabet fini, $\delta \subseteq \mathcal{Q} \times \Sigma \times \mathcal{Q}$ un ensemble de transitions, $\mathcal{Q}_0 \subseteq \mathcal{Q}$ un ensemble d'états initiaux, et $\mathcal{Q}_f \subseteq \mathcal{Q}$ un ensemble d'états finis.

La relation de transition $\rightarrow \subseteq \mathcal{Q} \times \Sigma^* \times \mathcal{Q}$ est définie comme étant la plus petite relation satisfaisant :

(1) $\forall q \in \mathcal{Q} : q \xrightarrow{\epsilon} q$,

(2) si $(q, \epsilon, q') \in \delta$, *alors* $q \xrightarrow{\epsilon} q'$,

(3) si $(q, a, q') \in \delta$, *alors* $q \xrightarrow{a} q'$ *et*

(4) si $q \xrightarrow{a} q'$ *et* $q' \xrightarrow{b} q''$, *alors* $q \xrightarrow{ab} q''$.

Un automate fini reconnait, à partir d'un alphabet, un ensemble de mots que l'on dénomme "Langage reconnu", dont la définition formelle est la suivante.

DÉFINITION 2.1.3 (Langage reconnu)
En utilisant les notations de la définition 2.1.2, le langage reconnu par un automate $\mathcal{A}$ sur un état $q \in \mathcal{Q}$ est défini par :

$$\mathcal{L}(\mathcal{A}, q) = \{w \mid \exists q' \in \mathcal{Q}_f \ tel \ que \ q \xrightarrow{w} q'\}.$$

Le langage d'un automate $\mathcal{A}$ est le langage reconnu par l'ensemble de ses états initiaux, i.e. $\mathcal{L}(\mathcal{A}) = \bigcup_{q_0 \in \mathcal{Q}_0} \mathcal{L}(\mathcal{A}, q_0)$.

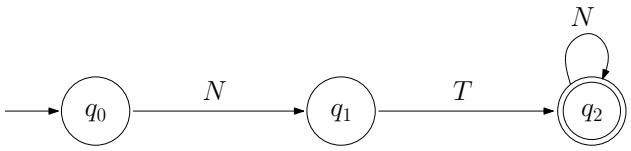


FIGURE 2.1 — Automate de mot représentant la configuration : le deuxième processus possède le jeton.

EXEMPLE *2.2*
Considérons le protocole de passage de jeton entre plusieurs processus placés les uns à la suite des autres. Chacun de ces processus peut être soit en repos, soit en section critique, selon qu'il possède ou non l'unique jeton disponible. Et chaque processus peut passer le jeton à son voisin de droite. Un état courant de ce protocole (pour un nombre de processus fini) peut alors être représenté par un mot sur l'alphabet $\{N, T\}$ où N représente un processus au repos, et T représente un processus possédant le jeton. Pour représenter une configuration (ou état) courante de ce protocole pour un nombre indéterminé (donc potentiellement infini) de processus, nous allons utiliser un automate de mot. En figure 2.1, est représenté l'ensemble des configurations dans laquelle le deuxième processus possède le jeton (quelque soit le nombre de processus).

On obtient alors l'automate suivant : $\mathcal{A} = \langle\{N, T\}, \{q_0, q_1, q_2\}, \{q_0\}, \{q_2\}, \delta\rangle$, avec $\delta = \{(q_0, N, q_1), (q_1, T, q_2), (q_2, N, q_2)\}$. Cet automate est également représenté en figure 2.1. On a alors $\mathcal{L}(\mathcal{A}) = \{NT(N)^*\}$. ◀

Définissons ensuite l'union puis l'intersection de deux automates de mots finis.

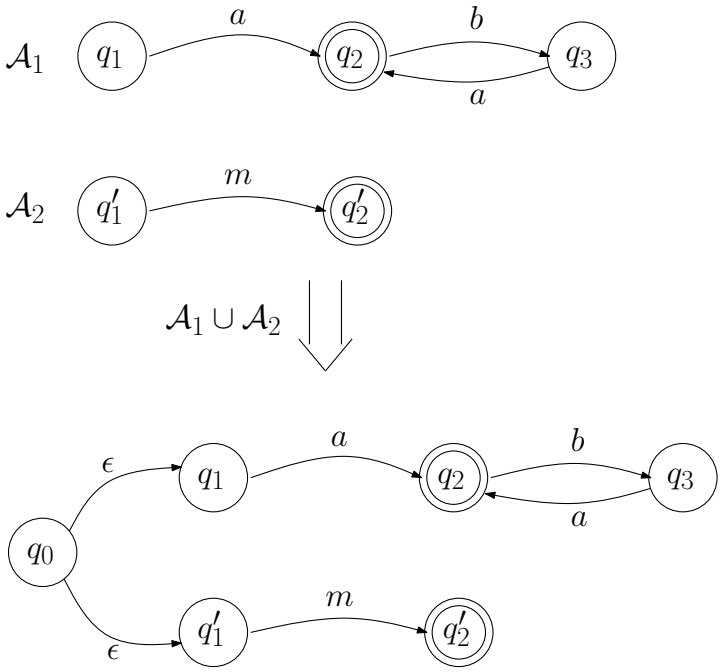


FIGURE 2.2 — Union de deux automates finis de mots.

DÉFINITION 2.1.4 (Union de deux automates finis)
Soient $\mathcal{A}_1 = \langle\Sigma, \mathcal{Q}_1, \mathcal{Q}_{0_1}, \mathcal{Q}_{f_1}, \delta_1\rangle$ *et* $\mathcal{A}_2 = \langle\Sigma, \mathcal{Q}_2, \mathcal{Q}_{0_2}\}, \mathcal{Q}_{f_2}, \delta_2\rangle$ *deux automates de mots*

finis tels que $\mathcal{Q}_1 \cap \mathcal{Q}_2 = \emptyset$*. Alors* $\mathcal{L}(\mathcal{A}_1 \cup \mathcal{A}_2) = \mathcal{L}(\mathcal{A}_1) + \mathcal{L}(\mathcal{A}_2)$*, et* $\mathcal{A}_1 \cup \mathcal{A}_2$ *peut se calculer de la manière suivante.*

$\mathcal{A}_1 \cup \mathcal{A}_2 = \mathcal{A}$*, avec* $\mathcal{A} = \langle \Sigma, \mathcal{Q}, \mathcal{Q}_0, \mathcal{Q}_f, \delta \rangle$*, où* $\mathcal{Q} = \mathcal{Q}_1 \cup \mathcal{Q}_2$*,* $\mathcal{Q}_0 = \{q_0\}$*, et* $\delta = \delta_1 \cup \delta_2 \cup \bigcup_{q \in \mathcal{Q}_{0_1} \cup \mathcal{Q}_{0_2}} \{(q_0, \epsilon, q)\}$*,* $\mathcal{Q}_f = \mathcal{Q}_{f_1} \cup \mathcal{Q}_{f_2}$*.*

Un exemple d'union est illustré en figure 2.2.

DÉFINITION 2.1.5 (Intersection de deux automates finis)
Soient $\mathcal{A}_1 = \langle \Sigma, \mathcal{Q}_1, \mathcal{Q}_{0_1}, \mathcal{Q}_{f_1}, \delta_1 \rangle$ *et* $\mathcal{A}_2 = \langle \Sigma, \mathcal{Q}_2, \mathcal{Q}_{0_1}, \mathcal{Q}_{f_2}, \delta_2 \rangle$ *deux automates de mots finis.*

Alors $\mathcal{A}_1 \cap \mathcal{A}_2 = \mathcal{A}$*, avec* $\mathcal{A} = \langle \Sigma, \mathcal{Q}, \mathcal{Q}_0, \mathcal{Q}_f, \delta \rangle$*, où* $\mathcal{Q} = \mathcal{Q}_1 \times \mathcal{Q}_2$*,* $\mathcal{Q}_0 = \mathcal{Q}_{0_1} \times \mathcal{Q}_{0_2}$*,* $\mathcal{Q}_f = \mathcal{Q}_{f_1} \times \mathcal{Q}_{f_2}$ *et* $\delta = \{((q_1, q_1'), a, (q_2, q_2')) \in \mathcal{Q} \times \Sigma \times \mathcal{Q} | (q_1, a, q_2) \in \delta_1 \wedge (q_1', a, q_2') \in \delta_2\} \cup \{((q_1, q_1'), \epsilon, (q_2, q_2')) \in \mathcal{Q} \times \{\epsilon\} \times \mathcal{Q} | (q_1 = q_2 \wedge (q_1', \epsilon, q_2') \in \delta_2) \vee (q_1' = q_2' \wedge (q_1, \epsilon, q_2) \in \delta_1)\}$*.*

Un exemple d'intersection est illustré en figure 2.3.

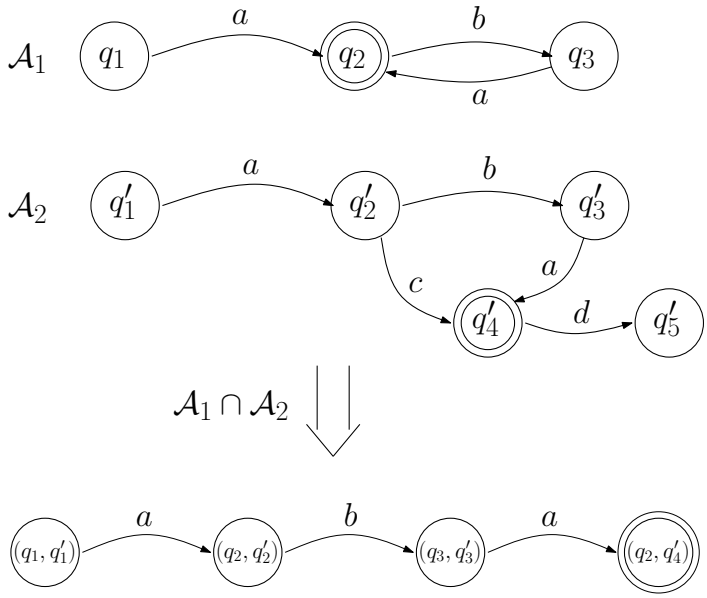


FIGURE 2.3 — Intersection de deux automates finis de mots.

Termes

Continuons maintenant avec les termes, construits depuis un alphabet de symboles fonctionnels et pouvant se représenter sous forme d'arbres.

DÉFINITION 2.1.6 (Alphabet de symboles fonctionnels)
Un alphabet $\mathcal{F}$ *est un alphabet de symboles fonctionnels si chacun de ses symboles possède une arité (i.e. un entier positif ou nul).* $\mathcal{F}^0$ *est l'ensemble des constantes (symboles d'arité 0) de* $\mathcal{F}$ *et* $\mathcal{F}^n$ *l'ensemble des symboles d'arité* n*.*

Un élément construit sur un alphabet $\mathcal{F}$ est appelé un terme, et peut se représenter sous forme d'arbre : les feuilles sont les symboles d'arité 0, et chaque symbole d'arité k possède k fils.

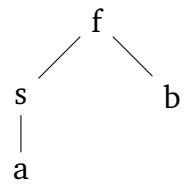


FIGURE 2.4 — Représentation du terme $f(s(a), b)$.

DÉFINITION 2.1.7 (Termes, termes clos, $\mathcal{T}(\mathcal{F}, \mathcal{X})$ et $\mathcal{T}(\mathcal{F})$)
Soient $\mathcal{F}$ un alphabet de symboles fonctionnels et $\mathcal{X}$ un ensemble fini de variables, alors $\mathcal{T}(\mathcal{F}, \mathcal{X})$ est l'ensemble des termes construit sur $\mathcal{F}$ et sur $\mathcal{X}$ et se définit récursivement comme suit :

(i). $X \in \mathcal{X} \Rightarrow X \in \mathcal{T}(\mathcal{F}, \mathcal{X})$

(ii). $a \in \mathcal{F}^0 \Rightarrow a \in \mathcal{T}(\mathcal{F}, \mathcal{X})$

(iii). $f \in \mathcal{F}^n$, $t_1, ..., t_n \in \mathcal{T}(\mathcal{F}, \mathcal{X}) \Rightarrow f(t_1, ..., t_n) \in \mathcal{T}(\mathcal{F}, \mathcal{X})$

*$\mathcal{T}(\mathcal{F})$ est l'ensemble des **termes clos** (termes sans variables).*

EXEMPLE *2.3*
Soient $\mathcal{F} = \{f_2, s_1, a_0, b_0\}$ un alphabet de symboles fonctionnels et $\mathcal{X} = \{x, y\}$ un ensemble de variables, on peut construire les termes suivants : $t_1 = f(s(a), b)$, $t_2 = f(x, b)$ et $t_3 = f(x, s(s(y)))$. On a alors $t_1 \in \mathcal{T}(\mathcal{F})$, t_2, $t_3 \in \mathcal{T}(\mathcal{F}, \mathcal{X})$. Le terme t_1 est représenté en figure 2.4. ◀

L'ensemble des variables d'un terme est noté $Var(t)$. Prenons par exemple les termes t_2 et t_3 de l'exemple 2.3. Alors $Var(t_2) = \{x\}$ et $Var(t_3) = \{x, y\}$.

DÉFINITION 2.1.8 (Substitution $\mathcal{X} \mapsto D$)
Soit $\mathcal{X}$ un ensemble de variables, soit D un ensemble et $\mathcal{F}$ un alphabet de symboles fonctionnels. Une substitution σ de $\mathcal{X}$ vers les éléments de D est une fonction $\sigma : \mathcal{X} \mapsto D$ qui peut être étendue de manière unique en un endomorphisme de $\mathcal{T}(\mathcal{F}, \mathcal{X})$ de cette manière. Pour tout $t \in \mathcal{T}(\mathcal{F}, \mathcal{X})$, $t\sigma$ est défini par :

1. *si $t = f(t_1, \ldots, t_n)$ alors $t\sigma = f(t_1\sigma, \ldots, t_n\sigma)$, avec $t, t_1, \ldots, t_n \in \mathcal{T}(\mathcal{F}, \mathcal{X})$, $f \in \mathcal{F}^n$,*
2. *si $t = a \in \mathcal{F}^0$ alors $t\sigma = t$,*
3. *si $t = x \in \mathcal{X}$ alors $t\sigma = \sigma(x)$.*

EXEMPLE *2.4*
Soit $D = \mathcal{T}(\mathcal{F})$ et σ une substitution $\mathcal{X} \mapsto \mathcal{T}(\mathcal{F})$ telle que $\sigma = \{x \mapsto g(a), y \mapsto h(h(b))\}$. Alors $f(x, y)\sigma = f(g(a), h(h(b)))\}$. ◀

DÉFINITION 2.1.9 (Position d'un terme, $Pos(t)$)
Soit $\mathcal{X}$ un ensemble de variables. Une position p pour un terme t est un mot sur $\mathbb{N}$. La séquence vide, notée ϵ, représente la position racine. L'ensemble $Pos(t)$ des positions d'un terme est défini de manière inductive par :

- $Pos(t) = \{\epsilon\}$ *si* $t \in \mathcal{X} \cup \mathcal{F}^0$
- $Pos(f(t_1, \ldots, t_n)) = \{\epsilon\} \cup \{i.p \mid 1 \leq i \leq n \text{ et } p \in Pos(t_i)\}$.

EXEMPLE *2.5*
Soit $t = f(a, f(x, f(a, b)))$, $Pos(t) = \{\epsilon, 1, 2, 2.1, 2.2, 2.2.1, 2.2.2\}$. ◀

DÉFINITION 2.1.10 (Sous-terme à une position p, $t|_p$)
Si $p \in Pos(t)$, alors on note $t|_p$ le sous-terme de t à la position p.

EXEMPLE *2.6*
Soit $t = f(a, f(x, f(a, b)))$, $t|_{2.2} = f(a, b)$. ◀

DÉFINITION 2.1.11 (Remplacement de terme, $t\left[s\right]_p$)
On note $t[s]_p$ le terme obtenu par remplacement du sous-terme $t|_p$ (de la position p) par le terme s.

EXEMPLE *2.7*
Soit $t = f(a, f(x, f(a, b)))$, $t[y]_{2.2} = f(a, f(x, y))$. ◀

Systèmes de réécriture

Définissons maintenant les systèmes de réécriture et certaines de leurs propriétés.

DÉFINITION 2.1.12 (Règle de réécriture)
Une règle de réécriture est une paire (l, r) notée $l \to r$, avec $l, r \in \mathcal{T}(\mathcal{F}, \mathcal{X})$, $l \notin \mathcal{X}$ et $Var(l) \supseteq Var(r)$.

DÉFINITION 2.1.13 (Système de réécriture)
Un système de réécriture $\mathcal{R}$ est un ensemble de règles de réécriture.

EXEMPLE *2.8*
Soient $\mathcal{F} = \{f_2, g_1, s_1, a_0, b_0\}$, $\mathcal{X} = \{x, y\}$ un ensemble de variables, on peut avoir le système de réécriture suivant : $\mathcal{R} = \{f(x, y) \to f(g(x), g(y)), g(x) \to s(x)\}$. ◀

DÉFINITION 2.1.14 (Équation)
Une équation est une paire de termes $(u, v) \in \mathcal{T}(\mathcal{F}, \mathcal{X}) \times \mathcal{T}(\mathcal{F}, \mathcal{X})$ notée $u = v$.

DÉFINITION 2.1.15 (Terme linéaire)
Un terme t est linéaire si chaque variable de $Var(t)$ apparait une seule fois dans t.

DÉFINITION 2.1.16 (Règle et système de réécriture linéaire, linéaire droit(e), linéaire gauche)
Une règle de réécriture $l \to r$ est linéaire gauche (respectivement linéaire droite) si l (respectivement r) est linéaire. Un système de réécriture est linéaire gauche (respectivement linéaire droit) si toutes ses règles de réécriture sont linéaires gauches (respectivement linéaires droites). Une règle (ou un système) de réécriture est linéaire si elle (il) est linéaire gauche et linéaire droit(e).

EXEMPLE *2.9*
Soient les systèmes de réécriture suivants : $\mathcal{R}_1 = \{f(x, y) \to f(g(x), g(x)), g(x) \to s(x)\}$, $\mathcal{R}_2 = \{f(x, x) \to f(g(x), g(x)), g(x) \to s(x)\}$. $\mathcal{R}_1$ est linéaire gauche, mais $\mathcal{R}_2$ ne l'est pas, car le membre gauche de sa première règle de réécriture contient deux fois la variable x. ◀

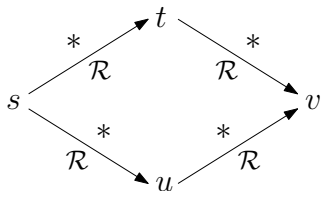


FIGURE 2.5 — Confluence d'un système de réécriture $\mathcal{R}$.

DÉFINITION 2.1.17 (Équation linéaire et ensemble d'équations linéaire)
Une équation $s = t$ est linéaire si les termes s et t sont linéaires. Un ensemble d'équations est linéaire si toutes ses équations sont linéaires.

La relation de réécriture est notée $\rightarrow_{\mathcal{R}}$ et est définie comme suit.

DÉFINITION 2.1.18 ($\rightarrow_{\mathcal{R}}$)
Soient $t_1, t_2 \in \mathcal{T}(\mathcal{F})$ deux termes clos, $\mathcal{R}$ un système de réécriture et $\mathcal{X}$ l'ensemble des variables de $\mathcal{R}$. On a $t_1 \rightarrow_{\mathcal{R}} t_2$ (le terme t_1 peut être réécrit en t_2) s'il existe une position $p \in Pos(t_1)$, une règle de réécriture $l \rightarrow r \in \mathcal{R}$ et une substitution $\sigma : \mathcal{X} \mapsto \mathcal{T}(\mathcal{F})$ telles que $t_1|_p = l\sigma$ et $t_2 = t_1[r\sigma]_p$

La clôture réflexive transitive de $\rightarrow_{\mathcal{R}}$ est notée $\rightarrow^*_{\mathcal{R}}$.

Nous allons maintenant définir ce qu'est un système de réécriture terminant.

DÉFINITION 2.1.19 (Terminaison)
Un système de réécriture $\mathcal{R}$ est dit terminant s'il n'y a pas de chaine infinie de réécriture $s_1 \rightarrow_{\mathcal{R}} s_2 \rightarrow_{\mathcal{R}} s_3 \rightarrow_{\mathcal{R}} \ldots$

EXEMPLE *2.10*
Soit $\mathcal{R} = \{s(x) \rightarrow s(s(x))\}$. Réécrire le terme $s(a)$, génère la chaine infinie de réécriture suivante :
$s(a) \rightarrow_{\mathcal{R}} s(s(a)) \rightarrow_{\mathcal{R}} s(s(s(a))) \rightarrow_{\mathcal{R}} s(s(s(s(a)))) \rightarrow_{\mathcal{R}} \ldots$ Le système de réécriture $\mathcal{R}$ n'est donc pas terminant. ◀

La réécriture est applicable à un ensemble de termes. Nous allons donc définir, pour un système de réécriture $\mathcal{R}$ et un ensemble de termes L donnés, l'ensemble des termes accessibles par à partir des termes de $\mathcal{L}$, noté $\mathcal{R}^*(\mathcal{L})$ et appelé les $\mathcal{R}$-descendants de L.

DÉFINITION 2.1.20 ($\mathcal{R}^*(\mathcal{L})$ ($\mathcal{R}$-descendants))
Soient $\mathcal{R}$ un système de réécriture et $\mathcal{L} \subseteq \mathcal{T}(\mathcal{F})$ un ensemble de termes. Alors l'ensemble des $\mathcal{R}$-descendants de $\mathcal{L}$, noté $\mathcal{R}^*(\mathcal{L})$, est l'ensemble $\{t \in \mathcal{T}(\mathcal{F}) \mid \exists s \in \mathcal{L}\ t.q.\ s \rightarrow^*_{\mathcal{R}} t\}$

DÉFINITION 2.1.21 (Confluence)
Un système de réécriture $\mathcal{R}$ est dit confluent si pour tout terme $s, t, u \in \mathcal{T}(\mathcal{F})$ tels que $s \rightarrow^*_{\mathcal{R}} t$ et $s \rightarrow^*_{\mathcal{R}} u$, il existe un terme $v \in \mathcal{T}(\mathcal{F})$ tel que $t \rightarrow^*_{\mathcal{R}} v$ et $u \rightarrow^*_{\mathcal{R}} v$.

Cette définition est illustrée en figure 2.5.

DÉFINITION 2.1.22 (E-équivalence ou égalité modulo E)
Soit E un ensemble d'équations, la relation de E-équivalence (notée $=_E$), ou encore l'égalité modulo E est définie comme suit :

- *soit deux termes clos $s, t \in \mathcal{T}(\mathcal{F})$, on dit que $s \approx t$ s'il existe une équation $u = v \in E$ et une substitution $\sigma : \mathcal{X} \mapsto \mathcal{T}(\mathcal{F})$ telles que $u\sigma = s$ et $v\sigma = t$.*
- *la relation d'équivalence $=_E \subseteq \mathcal{T}(\mathcal{F}) \times \mathcal{T}(\mathcal{F})$ est la plus petite congruence contenant la relation $\{(s, t) \in \mathcal{T}(\mathcal{F}) \times \mathcal{T}(\mathcal{F}) \mid s \approx t\}$.*

EXEMPLE *2.11*
Soient $E = \{s(x) = s(s(x))\}$, $s = s(s(g(a)))$ et $t = s(s(s(g(a))))$. Alors $s =_E t$ avec la substitution $\tau = x \mapsto s(g(a))$. ◀

Nous allons maintenant définir la relation de réécriture pouvant prendre en compte les égalités entre termes modulo un ensemble d'équations E.

DÉFINITION 2.1.23 (Réécriture modulo un ensemble d'équation)
Soient $\mathcal{R}$ un système de réécriture, E un ensemble d'équations et deux termes clos $s, t \in \mathcal{T}(\mathcal{F})$. Alors $s \to_{\mathcal{R}/E} t$ si $\exists s', t' \in \mathcal{T}(\mathcal{F})$ tels que $s =_E s' \to_{\mathcal{R}} t' =_E t$.

La clôture réflexive transitive de $\to_{\mathcal{R}/E}$ est notée $\to^*_{\mathcal{R}/E}$.

Et enfin, nous définissons l'ensemble des termes accessibles à partir d'un ensemble de termes $\mathcal{L}$, pour un système de réécriture $\mathcal{R}$ et un ensemble E d'équations donnés.

DÉFINITION 2.1.24 ($\mathcal{R}/E$-descendants)
Soient $\mathcal{R}$ un système de réécriture et $\mathcal{L} \subseteq \mathcal{T}(\mathcal{F})$ un ensemble de termes. Alors l'ensemble des $\mathcal{R}/E$-descendants de $\mathcal{L}$ (descendants modulo E), noté $\mathcal{R}^*_E(\mathcal{L})$, est l'ensemble $\{t \in \mathcal{T}(\mathcal{F}) \mid \exists s \in \mathcal{L}\ t.q.\ s \to^*_{\mathcal{R}/E} t\}$.

2.2 Automates d'arbres

Définissons maintenant les automates d'arbres, dont le langage est un ensemble (possiblement infini) de termes. Dans la modélisation d'un système informatique par un automate d'arbres, un terme permet de représenter une configuration (ou état) du système, et un automate d'arbres permet de représenter un ensemble de configurations (possiblement infini). Soient $\mathcal{F}$ un alphabet de symboles fonctionnels et $\mathcal{Q}$ un ensemble fini de symboles d'arité 0, appelés états, tels que $\mathcal{Q} \cap \mathcal{F} = \emptyset$. $\mathcal{T}(\mathcal{F} \cup \mathcal{Q})$ est l'ensemble des configurations sur $\mathcal{F}$. Il est défini inductivement comme suit.

DÉFINITION 2.2.1 (Ensemble de configurations ($\mathcal{T}(\mathcal{F} \cup \mathcal{Q})$))
Soient $\mathcal{F}$ un alphabet et $\mathcal{Q}$ un ensemble fini de symboles sur $\mathcal{F}^0$. On a alors :

(i). $\mathcal{Q} \subseteq \mathcal{T}(\mathcal{F} \cup \mathcal{Q})$,

(ii). Si $f \in \mathcal{F}^n$ et $s_1, ..., s_n \in \mathcal{T}(\mathcal{F} \cup \mathcal{Q})$, alors $f(s_1, ..., s_n) \in \mathcal{T}(\mathcal{F} \cup \mathcal{Q})$.

Un automate d'arbres est composé de transitions normalisées, définies de la manière suivante.

DÉFINITION 2.2.2 (Transition, transition normalisée et ϵ-transition)
Une transition est une règle de réécriture $c \to q$, avec c une configuration, i.e. $c \in \mathcal{T}(\mathcal{F} \cup \mathcal{Q})$ et $q \in \mathcal{Q}$. Une transition normalisée est une transition $c \to q$ de la forme suivante :

(i). $a \to q$, *avec* $a \in \mathcal{F}^0$, *et* $q \in \mathcal{Q}$, *ou*

(ii). $f(q_1, ..., q_n) \to q$, *avec* $f \in \mathcal{F}^n$, *et* q, q_1, ..., $q_n \in \mathcal{Q}$, *ou*

(iii). $q \to q'$, *avec* $q, q' \in \mathcal{Q}$. *Ce type de transitions est appelé une ϵ-transition.*

EXEMPLE *2.12*
Soient $\mathcal{F} = \{f_2, s_1, a_0\}$, et q, q_1, q_2 trois états. Alors la transition $f(q_1, q_2) \to q$ est normalisée. En revanche, la transition $f(g(a), q_2) \to q$ n'est pas normalisée, car la racine du terme (ici f) possède des fils qui ne sont pas des états, mais d'autres symboles de l'alphabet (ici g). La transition $q_1 \to q_2$ est une ϵ-transition. ◀

DÉFINITION 2.2.3 (Automate d'arbres)
Un automate d'arbres $\mathcal{A}$ est un tuple $\langle \mathcal{F}, \mathcal{Q}, \mathcal{Q}_f, \Delta \rangle$ avec $\mathcal{F}$ un alphabet de symboles fonctionnels, $\mathcal{Q}$ un ensemble fini d'états, $\mathcal{Q}_f \subseteq \mathcal{Q}$ l'ensemble des états finaux, et Δ un ensemble de transitions normalisées.

Définissons maintenant la relation de réécriture sur $\mathcal{T}(\mathcal{F} \cup \mathcal{Q})$ induite par les transitions de l'automate $\mathcal{A}$ (l'ensemble Δ). Cette relation est notée $\to_\Delta$.

DÉFINITION 2.2.4 ($\to_\Delta$)
Soient $t_1, t_2 \in \mathcal{T}(\mathcal{F} \cup \mathcal{Q})$ et un automate $\mathcal{A} = \langle \mathcal{F}, \mathcal{Q}, \mathcal{Q}_f, \Delta \rangle$, on a $t_1 \to_\Delta t_2$ (le terme t_1 peut être réduit en t_2) s'il existe une position $p \in Pos(t_1)$ telle que :
- *Si $t_1|_p \in \mathcal{F}^0$ et $t_1|_p \to q \in \Delta$, alors $t_2 = t_1[q]_p$.*
- *Si $t_1|_p = f(t_1, \ldots, t_n)$ et $t_1 \to_\Delta s_1, \ldots, t_n \to_\Delta s_n$, alors $t_2 = t_1[f(s_1, \ldots, s_n)]_p$.*

La notation $\to_\Delta^*$ définit la fermeture réflexive transitive de $\to_\Delta$. Quand le contexte est clair, cette relation est parfois notée $\to_\mathcal{A}^*$.

Un automate d'arbres $\mathcal{A}$ est une structure qui représente un langage régulier $\mathcal{L}$, qui est un ensemble (possiblement infini) de termes sur un alphabet $\mathcal{F}$. Définissons donc le langage décrit par un automate, que l'on note $\mathcal{L}(\mathcal{A})$.

DÉFINITION 2.2.5 ($\mathcal{L}(\mathcal{A})$)
Le langage d'un $\mathcal{A} = \langle \mathcal{F}, \mathcal{Q}, \mathcal{Q}_f, \Delta \rangle$ sur un état $q \in \mathcal{Q}$ est défini par $\mathcal{L}(\mathcal{A}, q) = \{t \in \mathcal{T}(\mathcal{F}) \mid t \to_\Delta^* q\}$. Le langage d'un automate est défini par $\mathcal{L}(\mathcal{A}) = \{t \in \mathcal{L}(\mathcal{A}, q) \mid q \in \mathcal{Q}_f\}$.

EXEMPLE *2.13*
$\mathcal{A} = \langle \mathcal{F}, \mathcal{Q}, \mathcal{Q}_F, \Delta \rangle$ avec $\mathcal{Q} = \{q_0, q_1, q_2\}$, $\mathcal{F} = \{f_2, a_0, b_0\}$, $\mathcal{Q}_f = \{q_3, q_4\}$, et

$$\Delta = \{ \quad a \to q_0 \ (1)\ , b \to q_1 \ (2)\ , \quad s(q_0) \to q_2 \ (3)\ , s(q_1) \to q_3 \ (4)\ , \quad f(q_2, q_1) \to q_4 \ (5) \ \}$$

Alors $\mathcal{L}(\mathcal{A}) = \{f(s(a), b), s(b)\}$. ◀

Un terme t appartient donc au langage d'un automate $\mathcal{A}$ s'il peut être réduit en un état final en appliquant certaines transitions de $\mathcal{A}$, comme illustré sur la figure 2.6.

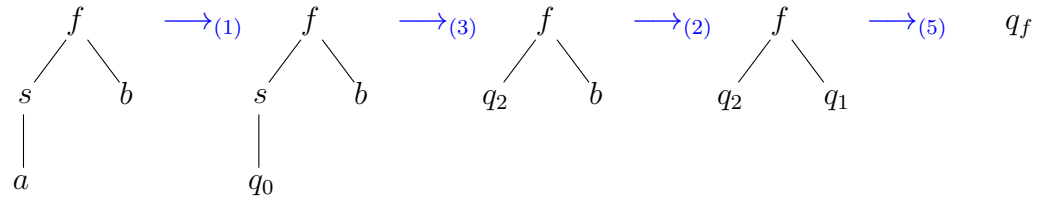


FIGURE 2.6 — Réduction du terme $f(s(a), b)$ en un état final, en utilisant les transitions de $\mathcal{A}$ de l'exemple 2.13. On a alors $f(s(a), b) \in \mathcal{L}(\mathcal{A})$.

Pour la suite, nous utiliserons des substitutions $\sigma : \mathcal{X} \mapsto \mathcal{Q}$ pour un ensemble de variables $\mathcal{X}$ et un ensemble d'états $\mathcal{Q}$ donnés (voir définition 2.1.8 où $D = \mathcal{Q}$).

EXEMPLE *2.14*
Soient q_3 et q_5 des états, et $\sigma = \{x \mapsto q_3, y \mapsto q_5\}$ une substitution. Alors si $t = f(g(x), h(h(y)))$, $t\sigma = f(g(q_3), h(h(q_5)))$. ◀

Les algorithmes des opérations sur les automates d'arbres (telles que l'union, l'intersection, la décision du vide, la déterminisation, etc.) sont sensiblement plus complexes, et ne seront donc pas détaillées ici par soucis de concision. De plus les idées de ces algorithmes sont similaires à ceux définis pour les automates de mots. Une description détaillée de ces algorithmes est disponible dans le chapitre 1 de [Comon et al., 2007].

2.3 Model-Checking régulier sur arbres

Comme expliqué brièvement dans l'introduction, cette thèse se place principalement dans le cadre de la vérification de propriétés de sûreté. Rappelons succinctement qu'une propriété de sûreté consiste à spécifier qu'un système ne peut jamais atteindre une mauvaise configuration. On appelle alors ceci un *problème d'atteignabilité*. Pour cela, il faut tout d'abord calculer l'ensemble des configurations atteignables par un système afin de vérifier qu'aucune mauvaise configuration n'y est présente. Le Model-Checking consiste alors à modéliser un système en séparant la modélisation de ses différentes configurations (ou **états**), et les actions permettant de passer d'une configuration à une autre (*i.e.*, ses **transitions**). Cette modélisation est appelée **système état-transition**.

DÉFINITION 2.3.1 (Système état-transition)
Un système état-transition est un tuple $\langle S, S_0, R\rangle$, avec :
- *S un ensemble (possiblement infini) d'états (ou configurations),*
- *S_0 un ensemble (possiblement infini) d'états initiaux, et*
- *$R \subseteq S \times S$ une relation d'atteignabilité (ou fonction de transition) permettant de décrire les transitions entre les états du système.*

Soient $\mathcal{T} = \langle S, S_0, R\rangle$ un système état-transition, et s, s' deux états de S. Si $(s, s') \in R$, alors cela signifie qu'il y a une transition de s jusqu'à s'. Alors on note $s \to_R s'$ si $(s, s') \in R$. Un état s' est atteignable depuis s s'il existe des états $s_0, s_1, \ldots, s_k \in S$ tels que $s \to_R s_0 \to_R s_1 \to_R \ldots \to_R s_k \to_R s'$. On dit alors que (s, s') appartient à la clôture réflexive transitive de R et ceci est noté $s \to_R^* s'$. Un état s est accessible par le système

s'il est atteignable depuis un état de S_0. L'ensemble des états accessibles de $\mathcal{T}$ est noté $S_R^{\mathcal{T}}$.

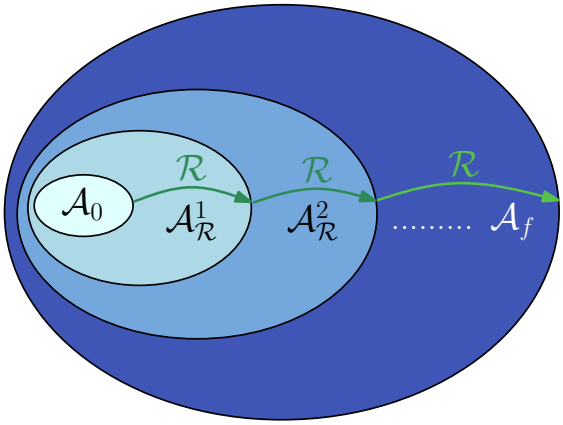


FIGURE 2.7 — Calcul de l'ensemble des configurations accessibles en *RTMC*.

Dans le cas de systèmes à ensembles d'états infinis, le premier problème difficile qui se pose est le suivant : comment représenter et manipuler des ensembles infinis d'états ? Pour résoudre ce problème récurrent en vérification, il n'existe que des solutions partielles. La solution proposée par le Model-Checking régulier est basée sur les langages réguliers de *mots* (*Regular Model-Checking : RMC* [Bouajjani et al., 2000]), ou sur les langages réguliers de *termes* (*Regular Tree Model-Checking : RTMC* [Abdulla et al., 2002]).

Le Model-Checking régulier sur *mots* ou arbres respecte le formalisme du système état-transition. En effet, on a alors :

- un *état* (ou configuration) du système est représentée par un *mot* ou un *terme*,
- un *ensemble d'états* (potentiellement infini) est représenté par un *automate de mots* ou un *automate d'arbres*,
- et la *fonction de transition* (ou fonction successeur) modélisant le comportement du système à vérifier (et permettant donc de passer d'une configuration à une autre, *i.e.*, d'un automate d'arbres à un autre) est représentée par un *transducteur* (dans le cas des mots), par un *système de réécriture* ou un *transducteur d'arbres* (dans le cas des arbres).

Nous utiliserons dans cette thèse les automates d'arbres comme représentation des ensembles d'états et les systèmes de réécritures en tant que fonction de transition. Le *RMC* (Model-Checking régulier sur mots) et les transducteurs d'arbres seront définis et comparés en section 3.2.1. Soit $\mathcal{A}_0$ l'automate d'arbres représentant l'ensemble des configurations initiales du système, $\mathcal{R}$ le système de réécriture représentant le comportement du système à modéliser et $\mathcal{F}$ l'alphabet du système. Dans le cadre du système état-transition, on a alors :

- $S_0 = \mathcal{L}(\mathcal{A}_0)$ (voir figure 2.7) l'ensemble des configurations initiales,
- $R = \rightarrow_{\mathcal{R}}$ (voir figure 2.7) la fonction de transition, et
- S représente tous les états du système donc l'ensemble des termes clos construits sur l'alphabets $\mathcal{F}$, soit l'ensemble $\mathcal{T}(\mathcal{F})$.
- $S_R^{\mathcal{T}} = \mathcal{R}^*(\mathcal{L}(\mathcal{A}_0))$ (voir définition 2.1.20) l'ensemble des configurations accessibles par ce système. L'automate "final" reconnaissant ce langage, ou une surapproximation de ce langage, est représenté par $\mathcal{A}_f$ sur la figure 2.7.

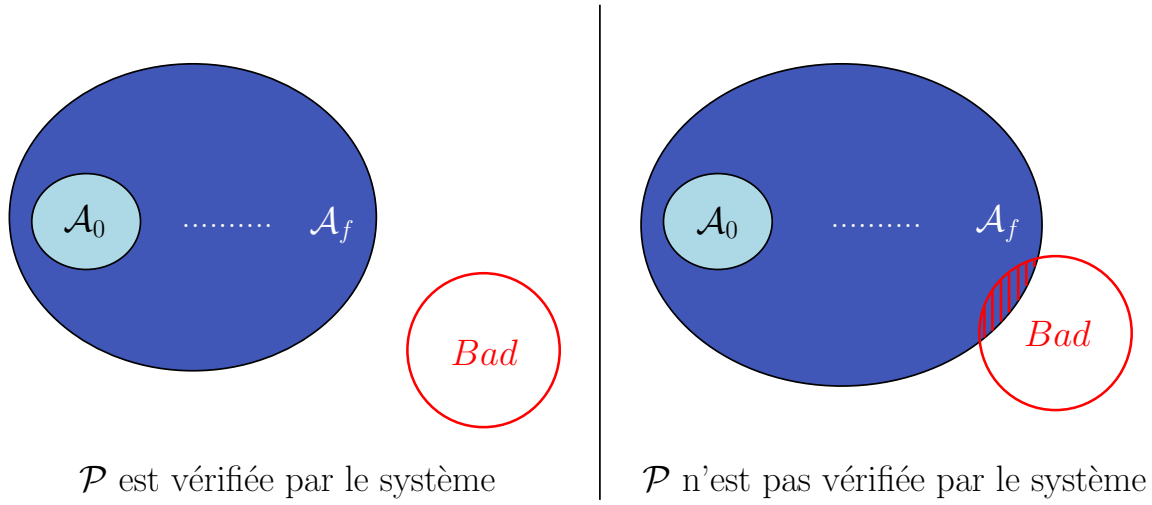


FIGURE 2.8 — Problème d'atteignabilité : l'ensemble des configurations accessibles d'un système possède-t-il des configurations interdites ?

Donnons ici formellement la définition de l'ensemble des configurations accessibles.

DÉFINITION 2.3.2 (Ensemble ou sur-ensemble des configurations accessibles $\mathcal{L}(\mathcal{A}_f)$)
Soient $\mathcal{A}_0$ un automate d'arbres et $\mathcal{R}$ un système de réécriture. Un automate $\mathcal{A}_f$ dont le langage contient l'ensemble des configurations accessibles depuis $\mathcal{A}_0$ ou une sur-approximation de cet ensemble, satisfait : $\mathcal{L}(\mathcal{A}_f) \supseteq \mathcal{R}^*(\mathcal{L}(\mathcal{A}_0))$.

Nous remarquons que le langage d'un tel automate contient le langage de tous les automates calculés précédemment grâce à la fonction de transition, soit $\mathcal{A}^1_{\mathcal{R}}$, $\mathcal{A}^2_{\mathcal{R}}$, $\mathcal{A}^3_{\mathcal{R}}$, ... (voir figure 2.7). Il en est de même pour chaque automate, à chaque étape du calcul. On a alors $\mathcal{L}(\mathcal{A}_0) \subseteq \mathcal{L}(\mathcal{A}^1_{\mathcal{R}}) \subseteq \ldots \subseteq \mathcal{L}(\mathcal{A}_f)$.

Pour prouver des propriété de sûreté et ainsi répondre au problème d'atteignabilité, il faut alors vérifier qu'aucune mauvaise configuration n'appartient au langage de l'automate $\mathcal{A}_f$. Dans le cadre de cette thèse, un problème d'atteignabilité est alors défini formellement comme suit.

DÉFINITION 2.3.3 (Problème d'atteignabilité)
Soient $\mathcal{A}_0$ l'automate représentant l'ensemble des configurations initiales d'un système $\mathcal{T}$, et $\mathcal{R}$ un système de réécriture représentant le comportement de $\mathcal{T}$. Alors une propriété de sûreté $\mathcal{P}$ consiste à vérifier qu'un ensemble de configurations donné n'est jamais accessible par $\mathcal{T}$. L'ensemble de configurations interdites est ici modélisé par un ensemble de termes Bad. Si aucun terme $t \in Bad$ n'est atteignable par $\mathcal{T}$, alors $\mathcal{T}$ vérifie la propriété de sûreté $\mathcal{P}$. On a alors :

$$\mathcal{T} \text{ vérifie } \mathcal{P} \iff \mathcal{R}^*(\mathcal{L}(\mathcal{A}_0)) \cap Bad = \emptyset$$

Le problème d'atteignabilité est résumé sur la figure 2.8, où l'ensemble des configurations accessibles est représenté par $\mathcal{A}_f$, et l'ensemble des configurations interdites est représenté par Bad.

Il faut alors parvenir à calculer cet ensemble de configurations accessibles ou une sur-approximation de cet ensemble, soit ici l'automate $\mathcal{A}_f$, en passant par des calculs intermédiaires, *i.e.* des automates intermédiaires (voir $\mathcal{A}^1_{\mathcal{R}}, \mathcal{A}^2_{\mathcal{R}}, \ldots$ sur la figure 2.7).

Ce calcul est le but de l'algorithme de complétion que nous allons décrire en section suivante.

Il est intéressant de noter que la classe des langages reconnaissables par automates d'arbres est close par intersection ($\cap$) et cette opération s'effectue en temps *polynomial* pour deux automates. En effet, l'intersection entre deux automates $\mathcal{A}_1$ et $\mathcal{A}_2$ possède une complexité en $O(||\mathcal{A}_1|| \times ||\mathcal{A}_2||)$ (où $||\mathcal{A}_i||$ est la taille de l'automate $\mathcal{A}_i$ [Comon et al., 2007]). Par ailleurs, la décision du vide, consistant à déterminer si le langage d'un automate d'arbres est vide ou non, est une opération décidable s'effectuant en temps *linéaire*. Les automates d'arbres sont donc tout à fait adaptés pour la vérification de propriété de sûreté, puisqu'il suffit de vérifier que $\mathcal{L}(\mathcal{A}_f) \cap Bad = \emptyset$, si Bad est un ensemble de termes régulier.

2.4 Algorithme de complétion d'automates d'arbres

Dans cette section, nous allons décrire la méthode utilisée et étendue tout au long de cette thèse, permettant de calculer l'ensemble des configurations accessibles d'un système (ou une sur-approximation de cet ensemble). L'algorithme de complétion permet en effet, à partir d'un automate d'arbres représentant l'ensemble des configurations initiales du système, d'appliquer successivement un système de réécriture jusqu'à atteindre un automate reconnaissant (une sur-approximation de) l'ensemble des configurations accessibles. Ici nous allons décrire l'algorithme de complétion définit dans [Genet et Rusu, 2010].

Chaque étape i de l'algorithme de complétion consiste à appliquer un système de réécriture $\mathcal{R}$ sur l'automate courant que l'on va noter $\mathcal{A}^i_{\mathcal{R}}$. Ceci va permettre d'enrichir le langage de $\mathcal{A}^i_{\mathcal{R}}$ (et ainsi ajouter les configurations découlant de ses actions ou comportement), et calculer l'automate *successeur* $\mathcal{A}^{i+1}_{\mathcal{R}}$. La *complétion* de $\mathcal{A}^i_{\mathcal{R}}$ se fait par l'ajout de nouvelles transitions.

L'algorithme de complétion consiste, depuis un automate initial $\mathcal{A}_0$, à itérer les étapes de calcul d'automates *successeurs* ($\mathcal{A}^1_{\mathcal{R}}, \mathcal{A}^2_{\mathcal{R}}, \ldots$: voir figure 2.7) jusqu'à obtenir un automate que l'on appelle *point-fixe*, *i.e.* un automate $\mathcal{A}^k_{\mathcal{R}}$ tel que $\mathcal{A}^k_{\mathcal{R}} = \mathcal{A}^{k+1}_{\mathcal{R}}$ pour $k \geq 0$. On dit aussi que le langage d'un automate point-fixe est $\mathcal{R}$-clos, *i.e.* qu'il respecte la définition suivante.

DÉFINITION 2.4.1 (Langage $\mathcal{R}$-clos)
Soient $\mathcal{R}$ un système de réécriture et $\mathcal{A}_0$ un automate d'arbres initial. Le langage d'un automate $\mathcal{A}_f$ est dit $\mathcal{R}$-clos si et seulement si, pour tout terme $s \in \mathcal{L}(\mathcal{A}_f)$, s'il existe un terme clos $t \in \mathcal{T}(\mathcal{F})$ tel que $s \to^*_{\mathcal{R}} t$, alors $t \in \mathcal{L}(\mathcal{A}_0)$.

Un automate est $\mathcal{R}$-clos s'il reconnait un langage $\mathcal{R}$-clos.

Le critère suivant est suffisant pour assurer que le langage de $\mathcal{A}_f$ est $\mathcal{R}$-clos : pour tout état q de $\mathcal{A}_f$, pour tout termes s, t tels que $s \to_{\mathcal{R}} t$,

$$s \to^*_{\mathcal{A}_f} q \;\Rightarrow\; t \to^*_{\mathcal{A}_f} q.$$

Ce critère est illustré en figure 2.9. Mais en pratique, comme il peut y avoir une infinité de termes s tels que $s \to_{\mathcal{R}} t$ et $s \to^*_{\mathcal{A}_f} q$, la $\mathcal{R}$-clôture est vérifiée de la manière suivante : un automate $\mathcal{A}_f$ est $\mathcal{R}$-clos si pour toute règle de réécriture $l \to r \in \mathcal{R}$, pour toute substitution $\sigma : \mathcal{X} \mapsto \mathcal{Q}$ telle que $l\sigma \to^*_{\mathcal{A}_f} q$, alors on a $r\sigma \to^*_{\mathcal{A}_f} q$.

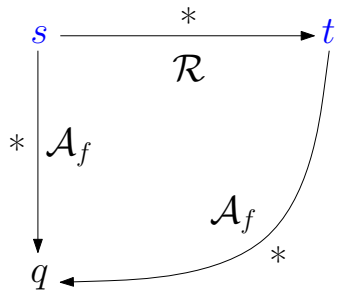


FIGURE 2.9 — $\mathcal{R}$-clôture de $\mathcal{A}_f$.

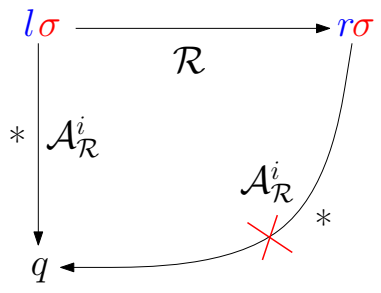


FIGURE 2.10 — Paire critique.

Comme il l'a été prouvé (notamment en Coq dans [Boyer et al., 2008]), $\mathcal{L}(\mathcal{A}_f) \supseteq \mathcal{R}^*(\mathcal{L}(\mathcal{A}_0))$ lorsque le langage de $\mathcal{A}_f$ est $\mathcal{R}$-clos et $\mathcal{L}(\mathcal{A}_f) \supseteq \mathcal{L}(\mathcal{A}_0)$. Un automate point-fixe possède nécessairement un langage $\mathcal{R}$-clos, mais un langage $\mathcal{R}$-clos n'est pas forcément reconnu par un automate point-fixe.Un automate point-fixe $\mathcal{R}$-clos sera noté $\mathcal{A}^*_{\mathcal{R}}$ par la suite.

2.4.1 Une étape de complétion

Le but ici est d'obtenir un automate point-fixe, donc $\mathcal{R}$-clos. D'un point de vue pratique et algorithmique, on détermine qu'un automate $\mathcal{A}_f$ est $\mathcal{R}$-clos si pour toute règle de réécriture $l \to r \in \mathcal{R}$, pour toute substitution $\sigma : \mathcal{X} \mapsto \mathcal{Q}$ telle que $l\sigma \to^*_{\mathcal{A}_f} q$, alors on a $r\sigma \to^*_{\mathcal{A}_f} q$.

Aussi, à chaque étape i de complétion (avec $\mathcal{A}^i_{\mathcal{R}}$ n'étant pas $\mathcal{R}$-clos), si pour une règle de réécriture $l \to r$ telle que $l\sigma \to^*_{\mathcal{A}_f} q$, si $r\sigma \not\to^*_{\mathcal{A}^i_{\mathcal{R}}} q$ alors on va enrichir le langage de $\mathcal{A}^i_{\mathcal{R}}$ afin que l'automate successeur $\mathcal{A}^{i+1}_{\mathcal{R}}$ puisse reconnaître le terme $r\sigma$ sur l'état q. Autrement dit, on aura alors $r\sigma \to^*_{\mathcal{A}^{i+1}_{\mathcal{R}}} q$ après le calcul de complétion.

Pour pouvoir ajouter tous les $r\sigma$ tels que $l \to r \in \mathcal{R}$, $l\sigma \to^*_{\mathcal{A}_f} q$, et $r\sigma \not\to^*_{\mathcal{A}^i_{\mathcal{R}}} q$, il faut calculer l'ensemble des *paires critiques* de $\mathcal{A}^i_{\mathcal{R}}$ et $\mathcal{R}$.

DÉFINITION 2.4.2 (Paires critiques)
Soient $\mathcal{A}$ un automate d'arbres, $\mathcal{Q}$ son ensemble d'états, et $\mathcal{R}$ un système de réécriture. Une paire critique de $\mathcal{A}$ et $\mathcal{R}$ est une paire $\langle r\sigma, q\rangle$ telle que $l\sigma \to^*_{\mathcal{A}} q$ et $r\sigma \not\to^*_{\mathcal{A}} q$, avec $\sigma : \mathcal{X} \mapsto \mathcal{Q}$ une substitution, $l \to r \in \mathcal{R}$, et $q \in \mathcal{Q}$.

Notons $PC(\mathcal{A}, \mathcal{R})$ l'ensemble des paires critiques de $\mathcal{A}$ et $\mathcal{R}$, alors

$$PC(\mathcal{A}, \mathcal{R}) = \bigcup_{\forall l \to r \in \mathcal{R}, \forall \sigma : \mathcal{X} \mapsto \mathcal{Q}, \forall q \in \mathcal{Q}} \{\langle r\sigma, q\rangle \mid l\sigma \to^*_{\mathcal{A}} q \text{ et } r\sigma \not\to^*_{\mathcal{A}} q\}$$

Cette définition est illustrée en figure 2.10.

EXEMPLE *2.15*
Soient $\mathcal{R} = \{s(x) \to s(s(x))$ et $\mathcal{A}$ un automate d'arbres possédant l'ensemble de transitions $\Delta = \{a \to q_0, s(q_0) \to q_1\}$. Avec la substitution $\sigma = \{x \mapsto q_0\}$, on remarque que $s(q_0) \to^*_{\mathcal{A}} q_1$, mais $s(s(q_0)) \not\to^*_{\mathcal{A}} q_1$. Alors $PC(\mathcal{A}, \mathcal{R}) = \{\langle s(s(q_0)), q_1\rangle\}$. ◀

Nous avons vu que chaque de étape de complétion i consistait à enrichir le langage de l'automate $A^i_{\mathcal{R}}$ afin que $r\sigma^*_{\mathcal{A}^{i+1}_{\mathcal{R}}} \to q$. Cela se fait en ajoutant toutes les transitions nécessaires à l'automate successeur $\mathcal{A}^{i+1}_{\mathcal{R}}$ pour toute paire critique $\langle r\sigma, q\rangle$ de

$PC(\mathcal{A}^i_{\mathcal{R}}, \mathcal{R})$. Cette étape est appelée *résolution des paires critiques* et sera détaillée dans le paragraphe "Résolution des paires critiques". Par exemple, la résolution des paires critiques de l'exemple 2.15 consistera donc à ajouter les transitions nécessaires pour avoir $s(s(q_0)) \to^* q_1$ dans l'automate successeur.

Pour l'instant, précisons tout d'abord la manière dont est calculé l'ensemble des paires critiques. Pour calculer l'ensemble des paires critiques $\langle r\sigma, q\rangle$, il faut calculer l'ensemble des substitutions σ telles que $l\sigma \to^*_{\mathcal{A}} q$ pour tout $l \to r \in \mathcal{R}$, et pour tout $q \in \mathcal{A}$ (voir définition 2.4.2). Le calcul de cet ensemble de substitutions nécessite un algorithme dit de *filtrage*, qui fait l'objet du paragraphe suivant.

Algorithme de filtrage

Un algorithme de filtrage permet, pour un système de réécriture $\mathcal{R}$ et un automate d'arbres $\mathcal{A}$ donnés, de calculer toutes les substitutions σ telles que $l\sigma \to^*_{\mathcal{A}} q$ pour tout $l \to r \in \mathcal{R}$, et pour tout $q \in \mathcal{A}$. Cet algorithme est défini en utilisant des règles de déduction sur des formules spécifiques dédiées, appelées *problèmes de filtrage*.

DÉFINITION 2.4.3 (Problème de filtrage)
Un problème de filtrage est une formule logique du premier ordre sans quantificateur où sont reliés, par les connecteurs $\oplus$ et $\otimes$, différents littéraux suivants : $\top, \bot, s, c$, avec $s \in \mathcal{T}(\mathcal{F}, \mathcal{X})$ un terme et $c \in \mathcal{T}(\mathcal{F} \cup \mathcal{Q})$ une configuration.

DÉFINITION 2.4.4 (Solutions d'un problème de filtrage)
Soient ϕ, ϕ_1, ϕ_2 des problèmes de filtrage, $s \in \mathcal{T}(\mathcal{F}, \mathcal{X})$ un terme, $c \in \mathcal{T}(\mathcal{F} \cup \mathcal{Q})$ une configuration et $\mathcal{A} = \langle \mathcal{F}, \mathcal{Q}, \mathcal{Q}_f, \Delta\rangle$ un automate d'arbres. Une solution d'un problème de filtrage est une substitution $\sigma : \mathcal{X} \mapsto \mathcal{Q}$ telle que :
- *si $\phi = s \trianglerighteq c$, alors $s\sigma \to_\Delta c$, et*
- *si $\phi = \phi_1 \otimes \phi_2$, alors σ est une solution de ϕ_1 et une solution de ϕ_2, et*
- *si $\phi = \phi_1 \oplus \phi_2$, alors σ est une solution de ϕ_1 ou une solution de ϕ_2.*

Donnons désormais l'algorithme de filtrage défini dans [Feuillade et al., 2004].

DÉFINITION 2.4.5
Soient $\mathcal{A} = \langle \mathcal{F}, \mathcal{Q}, \mathcal{Q}_f, \Delta\rangle$ un automate d'arbres, $f \in \mathcal{F}^n$, $g \in \mathcal{F}^m$, $q, q_1, \ldots, q_n \in \mathcal{Q}$, $q'_1, \ldots, q'_m \in \mathcal{Q}, s, s_1, \ldots, s_n \in \mathcal{T}(\mathcal{F}, \mathcal{X})$ et ϕ_1, ϕ_2, ϕ_3 des problèmes de filtrage non vides. L'algorithme de filtrage consiste à réduire tout problème de filtrage de la forme $s \trianglerighteq q$ à l'aide de l'ensemble de règles de déduction suivant.

(Configuration)

$$\frac{f(s_1, ..., s_n) \trianglerighteq q}{\bigoplus_{\forall f(q_1,...,q_n)\to q \in \Delta} f(s_1, ..., s_n) \trianglerighteq f(q_1, ..., q_n) \oplus \bot} \quad (f \in \mathcal{F}^n)$$

(Décomposition)

$$\frac{f(s_1, ..., s_n) \trianglerighteq f(q_1, ..., q_n)}{s_1 \trianglerighteq q_1 \otimes \ldots \otimes s_n \trianglerighteq q_n} \quad (f \in \mathcal{F}^n)$$

(Normalisation)

$$\frac{\phi_1 \otimes (\phi_2 \oplus \phi_3)}{(\phi_1 \otimes \phi_2) \oplus (\phi_1 \otimes \phi_2)} \quad , \quad \frac{\phi_1 \oplus \bot}{\phi_1} \quad et \quad \frac{\phi_1 \otimes \bot}{\bot}$$

EXEMPLE *2.16*
Soit $\mathcal{A} = \langle \{f_2, g_1, a_0, b_0\}, \{q_0, q_1, q_2\}, \{q_1\}, \Delta \rangle$, avec Δ l'ensemble de transitions suivant :

$$\begin{aligned} \Delta = \{ \quad & a \to q_0, b \to q_2, \\ & g(q_0) \to q_0, \\ & f(q_0, q_2) \to q_1, f(q_2, q_0) \to q_1 \ \}. \end{aligned}$$

Soit $\mathcal{R} = \{f(g(x), y) \to g(f(x), y)\}$. Si on applique l'algorithme de filtrage sur $f(g(x), y) \trianglerighteq q_1$, on obtient alors les règles de déduction suivantes.

$$\begin{array}{rcl} \text{(Configuration)} & \dfrac{f(g(x), y) \trianglerighteq q_1}{f(g(x), y) \trianglerighteq f(q_0, q_2) \ \oplus \ f(g(x), y) \trianglerighteq f(q_2, q_0)} & (f(q_0) \to q_1, \\ & & f(q_2) \to q_1 \in \Delta) \\ \text{(Décomposition)} & \dfrac{}{(g(x) \trianglerighteq q_0 \otimes y \trianglerighteq q_2) \ \oplus \ (g(x) \trianglerighteq q_2 \otimes y \trianglerighteq q_0)} & \\ \text{(Configuration)} & \dfrac{}{(g(x) \trianglerighteq g(q_0) \otimes y \trianglerighteq q_2) \ \oplus \ (\bot \otimes y \trianglerighteq q_0)} & (g(q_0) \to q_0 \in \Delta) \\ \text{(Décomposition)} & \dfrac{}{x \trianglerighteq q_0 \otimes y \trianglerighteq q_2} & \end{array}$$

La substitution $\sigma = \{x \mapsto q_0, y \mapsto q_2\}$ est donc une solution au problème de filtrage $f(g(x), y) \trianglerighteq q_1$. On peut alors déduire que $f(g(q_0), q_2) \to^*_\Delta q_1$.

◀

EXEMPLE *2.17*
Soit $\mathcal{A}_0 = \langle \mathcal{F}, \mathcal{Q}_0, \mathcal{Q}_{f_0}, \Delta_0 \rangle$, avec $\mathcal{F} = \{h_1, s_1, i_0, j_0\}$,$\mathcal{Q}_0 = \{q_0, q_1, q_2, q_3\}$, $\mathcal{Q}_{f_0} = \{q_3\}$, et Δ_0 l'ensemble de transitions suivant :

$$\begin{aligned} \Delta_0 = \{ \quad & i \to q_0, j \to q_1, \\ & s(q_0) \to q_2, s(q_1) \to q_2, \\ & h(q_2) \to q_3 \ \}. \end{aligned}$$

Soit $\mathcal{R} = \{h(s(x)) \to h(s(s(s(x)))\}$. Si l'on applique l'algorithme de filtrage sur $h(s(x)) \trianglerighteq q_3$, on obtient alors les règles de déduction suivantes.

$$\begin{array}{rcl} \text{(Configuration)} & \dfrac{h(s(x)) \trianglerighteq q_3}{h(s(x)) \trianglerighteq h(q_2)} & (h(q_2) \to q_3 \in \Delta_0) \\ \text{(Décomposition)} & \dfrac{}{(s(x) \trianglerighteq q_2)} & \\ \text{(Configuration)} & \dfrac{}{s(x) \trianglerighteq s(q_0) \ \oplus \ s(x) \trianglerighteq s(q_1)} & (s(q_0) \to q_2, s(q_1) \to q_2 \in \Delta_0) \\ \text{(Décomposition)} & \dfrac{}{x \trianglerighteq q_0 \ \oplus \ x \trianglerighteq q_1} & \end{array}$$

Les substitutions $\sigma_1 = \{x \mapsto q_0\}$ et $\sigma_2 = \{x \mapsto q_1\}$ sont deux solutions possibles au problème de filtrage $h(s(x)) \trianglerighteq q_3$. On peut alors déduire que $h(s(q_0)) \to^*_{\Delta_0} q_3$ et que $h(s(q_1)) \to^*_{\Delta_0} q_3$.

◀

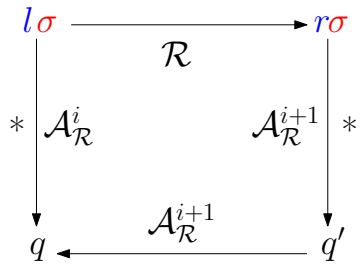


FIGURE 2.11 — Résolution des paires critiques.

Résolution des paires critiques

Maintenant que nous savons calculer l'ensemble des paires critiques d'un automate $\mathcal{A}$ et d'un système de réécriture $\mathcal{R}$ grâce à l'algorithme de filtrage, nous pouvons nous intéresser à la *résolution* de ces paires critiques, c'est-à-dire compléter l'automate courant pour enrichir son langage, dans le but d'obtenir un automate contenant tous les termes accessibles. Dans l'algorithme de complétion proposé par [Genet et Rusu, 2010], pour chaque paire critique $\langle r\sigma, q\rangle \in PC(\mathcal{A}, \mathcal{R})$, l'automate $\mathcal{A}^{i+1}_{\mathcal{R}}$ est construit en ajoutant $r\sigma \to^* q'$ et $q' \to q$ à l'automate $\mathcal{A}^i_{\mathcal{R}}$, afin que $\mathcal{A}^{i+1}_{\mathcal{R}}$ reconnaisse $r\sigma$ en q, *i.e.* $r\sigma \to^*_{\mathcal{A}^{i+1}_{\mathcal{R}}} q$. L'état q' est alors un *nouvel* état, non présent dans $\mathcal{A}^i_{\mathcal{R}}$. L'$\epsilon$-transition $q' \to q$ est ajoutée afin de garder un historique des différentes étapes de complétion dans l'ensemble des transitions, et permet plus de précision lors de l'étape d'approximation (voir section *Abstraction équationnelle*). La résolution des paires critiques est résumée en figure 2.11.

Nous allons désormais nous intéresser à la manière dont va être ajouté l'ensemble de transitions nécessaire pour avoir $r\sigma \to^* q'$ dans l'automate successeur. En effet, rappelons qu'un automate doit être composé de *transitions normalisées* (voir définition 2.2.2). Si $r\sigma \to q'$ n'est pas une transition normalisée (comme par exemple $f(g(q)) \to q'$), elle ne peut pas être ajoutée telle quelle dans l'ensemble de transitions d'un automate. La normalisation d'une transition consiste à la décomposer en plusieurs transitions normalisées, et ceci nécessite souvent l'utilisation de nouveaux états, comme nous allons le voir dans la définition suivante.

DÉFINITION 2.4.6 (Normalisation)
Soient $s \in \mathcal{T}(\mathcal{F} \cup \mathcal{Q})$, $q \in \mathcal{Q}$ et $\mathcal{A} = \langle \mathcal{F}, \mathcal{Q}, \mathcal{Q}_f, \Delta\rangle$ un automate d'arbres. Un nouvel état est un état n'apparaissant pas dans $\mathcal{Q}$. L'opération $Norm(s \to q)$ retourne l'ensemble des transitions normalisées déduites de s et est définie inductivement par :

1. *si $s \in \mathcal{F}^0 \cup \mathcal{Q}$, alors $Norm(s \to q) = \{s \to q\}$.*
2. *si $s = f(t_1, \ldots, t_n)$, avec $f \in \mathcal{F}^n$, alors $Norm(s \to q) = \{f(q'_1, \ldots, q'_n) \to q\} \cup Norm(t_1 \to q'_1) \cup \ldots \cup Norm(t_n \to q'_n)$, où, pour tout $i \in [1, n]$, q'_i est :*
   - *soit la partie droite d'une transition de Δ telle que $t_i \to^*_\Delta q'_i$ (sans ϵ-transitions),*
   - *soit un nouvel état.*

Si plusieurs choix d'états peuvent être faits lors de la normalisation, même si l'automate après normalisation ne sera pas le même selon le choix effectué, tous les choix conviennent et donneront un résultat correct.

EXEMPLE *2.18*
Si nous reprenons l'automate $\mathcal{A}_0$ de l'exemple 2.17, $Norm(h(s(s(s(q_0))) \to q'_3) = \{s(q_0) \to q_2, s(q_2) \to q_4, s(q_4) \to q_5, h(q_5) \to q'_3\}$, avec q_4 et q_5 de nouveaux états, et q_2 un état existant car la transition $s(q_0) \to q_2$ est déjà présente dans Δ_0.

◀

Définition d'une étape de complétion

Nous avons désormais tous les éléments nécessaires pour définir une étape de complétion. Il est essentiel de préciser que pour l'algorithme de complétion et dans cette thèse en général, nous restreindrons la complétion à des systèmes de réécritures *linéaires gauches* (voir définition 2.1.16). En effet, le fait d'avoir plusieurs fois la même variable dans la partie gauche de la règle peut compliquer l'algorithme de complétion, notamment au niveau de l'algorithme de filtrage. Des solutions sont proposées dans [Boichut et al., 2009, Takai et al., 2000] pour enlever cette restriction.

DÉFINITION 2.4.7 (Une étape de complétion : calcul de l'automate successeur $\mathcal{A}_{\mathcal{R}}^{i+1} = \mathcal{C}_{\mathcal{R}}(\mathcal{A})$)
Soient $\mathcal{A} = \langle \mathcal{F}, \mathcal{Q}, \mathcal{Q}_f, \Delta \rangle$ un automate d'arbres, et $\mathcal{R}$ un système de réécriture linéaire gauche. L'automate d'arbres obtenu après une étape de complétion est noté $\mathcal{C}_{\mathcal{R}}(\mathcal{A}) = \langle \mathcal{F}, \mathcal{Q}', \mathcal{Q}'_f, \Delta' \rangle$, avec :

$$\Delta' = \Delta \cup \bigcup_{\langle r\sigma, q\rangle \in PC(\mathcal{A},\mathcal{R})} Norm(r\sigma \to q') \cup \{q' \to q\},$$

avec q' un nouvel état (chaque fois différent), $\mathcal{Q}'$ contenant tous les états de Δ', et pour tout $\langle r\sigma, q\rangle \in PC(\mathcal{A}, \mathcal{R})$, si $q \in \mathcal{Q}_f$, alors $q' \in \mathcal{Q}'_f$.

EXEMPLE *2.19*
Si on reprend les données de l'exemple 2.17, l'ensemble des substitutions calculées pour $\mathcal{A}_0$ et $\mathcal{R}$ sont $\sigma_1 = \{x \mapsto q_0\}$ et $\sigma_2 = \{x \mapsto q_1\}$. On a donc :

$$PC(\mathcal{A}_0, \mathcal{R}) = \{\langle h(s(s(s(q_0))), q_3\rangle, \langle h(s(s(s(q_1))), q_3\rangle\}.$$

D'après l'exemple 2.18, $Norm(h(s(s(s(q_0))) \to q'_3) = \{s(q_0) \to q_2, s(q_2) \to q_4, s(q_4) \to q_5, h(q_5) \to q'_3\}$. On a aussi $Norm(h(s(s(s(q_1))) \to q'_3) = \{s(q_1) \to q_2, s(q_2) \to q_4, s(q_4) \to q_5, h(q_5) \to q'_3\}$. Étant donné que les deux transitions $s(q_0) \to q_2$ et $s(q_1) \to q_2$ ont une partie droite commune (q_2), alors leur normalisation est la même. Comme ces transitions sont déjà présentes dans l'automate, on a alors $\mathcal{C}_{\mathcal{R}}(\mathcal{A}_0) = \langle \mathcal{F}, \mathcal{Q}_1, \mathcal{Q}_{f_1}, \Delta_1 \rangle$, avec :

$$\Delta_1 = \Delta_0 \cup \{ \begin{array}{l} s(q_2) \to q_4, s(q_4) \to q_5, \\ h(q_5) \to q'_3, q'_3 \to q_3 \ \}, \end{array}$$

et $\mathcal{Q}_1 = \mathcal{Q}_0 \cup \{q'_3, q_4, q_5\}$. ◀

2.4.2 Abstraction par équations

Limites du calcul de complétion

L'algorithme de complétion consiste à itérer les étapes de complétion jusqu'à tomber sur un point-fixe. Cependant, il est fréquent que ce calcul soit non terminant, excepté pour certaines classes de réécritures définies dans [Genet, 2009].

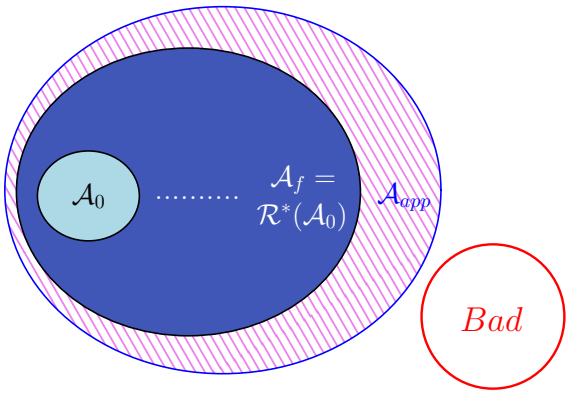


FIGURE 2.12 — Sur-approximation calculable de l'ensemble des termes accessibles.

EXEMPLE *2.20*
Soit $\mathcal{R} = \{s(x) \to s(s(x))\}$ et $\mathcal{A}$ l'automate d'arbres possédant l'ensemble de transitions suivant : $\Delta = \{a \to q_0, s(q_0) \to q_1\}$, avec q_1 l'état final. Le tableau suivant résume quelles paires critiques sont résolues et les transitions ajoutées par chaque étape de complétion pour obtenir l'automate $\mathcal{A}_i$:

$\mathcal{A}_0$:	$a \to q_0$,	$s(q_0) \to q_1$,	$\langle s(s(q_0)), q_1 \rangle$
$\mathcal{A}_1$:	$s(q_1) \to q_2$,	$q_2 \to q_1$,	$\langle s(s(q_1)), q_2 \rangle$
$\mathcal{A}_2$:	$s(q_2) \to q_3$,	$q_3 \to q_2$,	$\langle s(s(q_2)), q_3 \rangle$
$\mathcal{A}_3$:	$s(q_3) \to q_4$,	$q_4 \to q_3$,	$\langle s(s(q_3)), q_4 \rangle$
	...	...	...

On remarque ici facilement que le calcul de complétion est divergent, car le nouvel état introduit à chaque étape de complétion fournit une nouvelle paire critique avec la règle $s(x) \to s(s(x))$.

◀

De plus, certains langages représentant l'ensemble des configurations accessibles ne sont pas représentables par un automate d'arbres, donc non calculables avec l'algorithme de complétion. C'est le cas où les configurations accessibles sont représentées par un langage non-régulier [Boichut et Héam, 2008].

EXEMPLE *2.21*
Soient $\mathcal{R} = \{f(x, y) \to f(s(x), s(y))\}$ un système de réécriture et $\mathcal{A}$ un automate d'arbres possédant l'ensemble de transitions suivant : $\Delta = \{a \to q_0, f(q_0, q_0) \to q_1\}$. Soit q_1 l'état final. Alors le langage des configurations accessibles est $f(s^n(a), s^n(a))$ et ce langage n'est pas régulier : il n'est donc pas possible de le représenter par un automate d'arbres. ◀

Principe de la sur-approximation

Dans le cas où le calcul de complétion diverge, une sur-approximation de l'ensemble des configurations accessibles est calculée, comme illustré en figure 2.12.

Pour cela, plusieurs techniques d'abstractions sont possibles, et elles reposent toutes sur la fusion d'états considérés comme étant équivalents selon certains critères. La fu-

sion de deux états q et q' consiste à remplacer chaque occurrence de q par q' (ou inversement) dans l'automate d'arbres. Cette fusion consiste alors en un renommage d'état, dont la définition formelle est la suivante.

DÉFINITION 2.4.8 (Renommage d'un état dans un automate d'arbres)
Soient $\mathcal{Q}$ et $\mathcal{Q}'$ deux ensembles d'états, $\mathcal{A} = \langle \mathcal{F}, \mathcal{Q}, \mathcal{Q}_f, \Delta \rangle$ un automate d'arbres et $\alpha : \mathcal{Q} \mapsto \mathcal{Q}'$ une fonction. On note $\mathcal{A}\alpha$ l'automate d'arbres où chaque occurrence de q est remplacée par $\alpha(q)$ de manière récursive, dans $\mathcal{Q}, \mathcal{Q}_f$ et dans chaque transition de Δ.

S'il existe une bijection α telle que $\mathcal{A} = \mathcal{A}'\alpha$, alors $\mathcal{A}$ et $\mathcal{A}'$ sont dit *équivalents modulo renommage*.

EXEMPLE *2.22*
Reprenons l'exemple 2.20. Si on décide de fusionner q_2 et q_3 dans $\mathcal{A}_2$, alors l'automate $\mathcal{A}_2\{q_3 \mapsto q_2\}$ après renommage de q_3 en q_2 possède l'ensemble de transitions suivant :

$$\begin{aligned}\Delta_{\mathcal{A}_2\{q_3 \mapsto q_2\}} = \{ \quad & a \to q_0, s(q_0) \to q_1, \\ & s(q_1) \to q_2, q_2 \to q_1, \\ & s(q_2) \to q_2 \ \}\end{aligned}$$

On remarque ici qu'il n'y a plus de paires critiques non résolues. En effet, $s(q_2) \to^* q_2$ et $s(s(q_2)) \to^* q_2$. L'automate $\mathcal{A}_2\{q_3 \mapsto q_2\}$ est alors un point-fixe et le calcul de complétion termine.

◀

Abstraction par équations

Il existe plusieurs manières de calculer des classes d'équivalence entre les différents états de l'automate courant, pour déterminer quels états il faudra fusionner pour abstraire le calcul de complétion. La technique la plus récente, qui sera utilisée et étendue dans cette thèse, est la fusion d'état équivalents selon un ensemble d'équations. Nous reviendrons sur diverses autres méthodes dans le chapitre 3.

La fusion d'états équivalents selon un ensemble d'équations est une technique définie dans [Genet et Rusu, 2010] de la manière suivante.

DÉFINITION 2.4.9 (Abstraction par équations)
Soient $\mathcal{A} = \langle \mathcal{F}, \mathcal{Q}, \mathcal{Q}_f, \Delta \rangle$ un automate d'arbres et E un ensemble d'équations de la forme $u = v$ avec $u, v \in \mathcal{T}(\mathcal{F}, \mathcal{X})$.

*Pour chaque $u = v \in E$, pour chaque substitution $\sigma : \mathcal{X} \mapsto \mathcal{Q}$, pour chaque état q, q' tels que $u\sigma \to^*_\Delta q$ et $v\sigma \to^*_\Delta q'$ et $q \neq q'$, alors $\mathcal{A}$ peut être abstrait par $\mathcal{A}\{q' \mapsto q\}$, i.e. q et q' sont **fusionnés**. On note alors cette abstraction $\mathcal{A} \rightsquigarrow^!_E \mathcal{A}'$.*

Cette définition est illustrée en figure 2.13. Notons que pour calculer les substitutions telles que $u\sigma \to^* q$ et $v\sigma \to^* q'$, l'algorithme de filtrage décrit en section 2.4.1 est utilisé.

EXEMPLE *2.23*
Reprenons l'exemple 2.19. Rappelons que l'on a $\mathcal{R} = \{h(s(x)) \to h(s(s(s(x)))\}$, et l'ensemble de transitions suivant après une étape de complétion :

$$\begin{aligned}\Delta' = \{ \quad & i \to q_0, j \to q_1, s(q_0) \to q_2, s(q_1) \to q_2, \\ & h(q_2) \to q_3, s(q_2) \to q_4, s(q_4) \to q_5, \\ & h(q_5) \to q'_3, q'_3 \to q_3 \ \},\end{aligned}$$

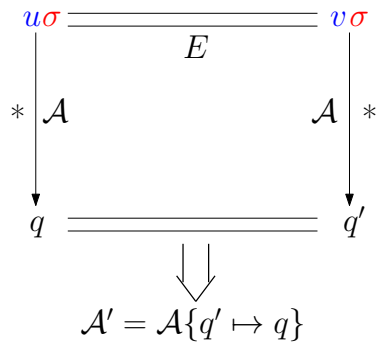


FIGURE 2.13 — Fusion par l'équation $u = v$.

Soit $E = \{s(x) = s(s(s(x)))\}$. On a alors $s(q_0) \to^*_\Delta q_2$ et $s(s(s(q_0))) \to^*_\Delta q_5$, mais également $s(q_1) \to^*_\Delta q_2$ et $s(s(s(q_1))) \to^*_\Delta q_5$. Les états q_2 et q_5 sont donc équivalents par E et peuvent être fusionnés. On obtient alors l'automate $\mathcal{A}'$ possédant l'ensemble de transitions suivant, avec $\mathcal{A}' = \mathcal{A}\{q_5 \mapsto q_2\}$.

$$\begin{aligned}\Delta' = \{ \quad & i \to q_0, j \to q_1, s(q_0) \to q_2, s(q_1) \to q_2,\\ & h(q_2) \to q_3, s(q_2) \to q_4, s(q_4) \to q_2,\\ & h(q_2) \to q'_3, q'_3 \to q_3 \ \},\end{aligned}$$

Ici on a $h(q_2) \to^*_\Delta q'_3$ et également $h(s(s(s(q_2))) \to^*_\Delta q'_3$. L'automate est donc $\mathcal{R}$-clos et point-fixe.

◀

2.4.3 Algorithme de complétion avec abstraction par équation

Nous possédons désormais tous les éléments pour définir l'algorithme de complétion, incluant l'abstraction par équation, que l'on trouve dans [Genet et Rusu, 2010]. Avant de définir formellement l'intégralité de cet algorithme, nous allons tout d'abord résumer le fonctionnement de cet algorithme de manière schématique et informelle.

Soit $\mathcal{R}$ un système de réécriture et E un ensemble d'équations.
Tant que $\mathcal{A}^i_{\mathcal{R},E} \neq \mathcal{A}^{i+1}_{\mathcal{R},E}$ (tant qu'il n'y a **pas de point-fixe**), faire :

1. Pour tout $l \to r$ de $\mathcal{R}$, appliquer l'**algorithme de substitution** pour trouver toutes substitutions σ telles que $l\sigma \to^*_{\mathcal{A}^i_{\mathcal{R},E}} q$ et $r\sigma \not\to^*_{\mathcal{A}^i_{\mathcal{R},E}} q$ avec q un état de $\mathcal{A}^i_{\mathcal{R}}$: *détection des **paires critiques*** $\langle r\sigma, q\rangle$.
2. Pour toute paire critique $\langle r\sigma, q\rangle$, calculer la **normalisation** des transitions $r\sigma \to q'$, avec q' un *nouvel* état.
3. **Ajouter les transitions** normalisées $Norm(r\sigma \to q')$ ainsi que $q' \to q$ à l'automate $\mathcal{A}^i_{\mathcal{R}}$. On obtient alors l'automate $\mathcal{C}_{\mathcal{R}}(\mathcal{A}^i_{\mathcal{R}})$: *résolution des paires critiques*.
4. Pour toute équation $u = v$ de E, calculer toutes les substitutions σ telles que $u\sigma \to^*_{\mathcal{C}_{\mathcal{R}}(\mathcal{A}^i_{\mathcal{R}})} q_1$ et $v\sigma \to^*_{\mathcal{C}_{\mathcal{R}}(\mathcal{A}^i_{\mathcal{R}})} q_2$, avec q_1, q_2 des états de $\mathcal{C}_{\mathcal{R}}(\mathcal{A}^i_{\mathcal{R}})$.
5. Pour toute paire d'états (q_1, q_2) équivalents par équation, **fusionner** les états q_1 et q_2 dans l'automate. On obtient alors l'automate $\mathcal{A}^{i+1}_{\mathcal{R},E}$.

Donnons maintenant la définition formelle de cet algorithme.

DÉFINITION 2.4.10 (Algorithme de complétion)
Soient $\mathcal{A}$ un automate d'arbres, $\mathcal{R}$ un système de réécriture et E un ensemble d'équations.
- $\mathcal{A}^0_{\mathcal{R},E} = \mathcal{A}$,
- *Répéter* $\mathcal{A}^{n+1}_{\mathcal{R},E} = \mathcal{A}'$ *avec* $\mathcal{C}_{\mathcal{R}}(\mathcal{A}^n_{\mathcal{R},E})) \rightsquigarrow^!_E \mathcal{A}'$,
- *Jusqu'à ce qu'un point-fixe* $\mathcal{A}^*_{\mathcal{R},E} = \mathcal{A}^k_{\mathcal{R},E} = \mathcal{A}^{k+1}_{\mathcal{R},E}$ *(avec* $k \in \mathbb{N}$*) soit atteint.*

EXEMPLE *2.24*
Déroulons l'intégralité de l'algorithme de complétion incluant l'approximation par équations sur un exemple, afin de vérifier une propriété sur le système modélisé.

Soient $\mathcal{R} = \{cons(x,y) \to cons(s(s(x)), cons(x,y))\}$ un système de réécriture, $E = \{s(x) = s(s(s(x)))\}$, et $\mathcal{A}_0$ l'automate d'arbres possédant l'ensemble de transitions $\Delta_0 = \{zero \to q_0, nil \to q_{nil}, cons(q_0, q_{nil}) \to q_{f_0}\}$, avec q_{f_0} l'état final. Autrement dit, $\mathcal{L}(\mathcal{A}_0) = \{cons(zero, nil)\}$. Les étapes de complétion sont résumées dans le tableau 2.1. Pour simplifier la présentation, les transitions communes ne sont pas répétées, *i.e.* $\mathcal{A}^i_{\mathcal{R},E}$ est supposé contenir toutes les transitions des automates $\mathcal{A}^{i-1}_{\mathcal{R},E}, \ldots, \mathcal{A}^0$.

L'automate $\mathcal{A}^1_{\mathcal{R},E}$ est directement le résultat $\mathcal{C}_{\mathcal{R}}(\mathcal{A}^0)$ car aucune équation ne peut être appliquée. L'automate $\mathcal{A}^2_{\mathcal{R},E}$ est obtenu en fusionnant q_1 avec q_3 et q_2 avec q_4, soit $\mathcal{A}^2_{\mathcal{R},E} = \mathcal{C}_{\mathcal{R}}(\mathcal{A}^1_{\mathcal{R},E})\{q_3 \mapsto q_1, q_4 \mapsto q_2\}$. Il n'y a plus de paires critiques pour l'automate $\mathcal{A}^2_{\mathcal{R},E}$, il s'agit donc d'un point-fixe et ainsi $\mathcal{L}(\mathcal{A}^2_{\mathcal{R},E}) \supseteq \mathcal{R}^*(\mathcal{L}(\mathcal{A}_0))$.

Si nous voulons vérifier la propriété "le système construit une liste de nombres pairs", alors l'ensemble de termes interdits Bad peut être l'automate représentant l'ensemble des listes contenant au moins un nombre impair : il s'agit alors de l'automate d'arbres $\mathcal{A}_{Bad}$, dont q_i est le seul état final, et qui possède l'ensemble de transitions suivant.

$$\begin{aligned}\Delta_{Bad} = \{ \quad & zero \to q'_0, nil \to q_p, \\ & s(q'_0) \to q'_1, s(q'_1) \to q'_0, \\ & cons(q'_0, q_p) \to q_p, cons(q'_1, q_p) \to q_i \\ & cons(q'_0, q_i) \to q_i, cons(q'_1, q_i) \to q_i \ \}\end{aligned}$$

Après calcul de $\mathcal{A}^2_{\mathcal{R},E} \cap \mathcal{A}_{Bad}$, on constate que $\mathcal{L}(\mathcal{A}^2_{\mathcal{R},E} \cap \mathcal{A}_{Bad}) = \emptyset$, et on peut donc déduire que $\mathcal{L}(\mathcal{R}^*(\mathcal{A}_0)) \cap Bad = \emptyset$. ◀

Donnons maintenant le théorème de complétude de cet algorithme, qui permet de garantir que tous les états accessibles du système sont inclus dans l'automate point-fixe calculé par complétion avec approximation. Les preuves peuvent être trouvées dans [Genet et Rusu, 2010, Genet, 2009].

THÉORÈME 2.4.11
Soient $\mathcal{A}$ un automate d'arbres, $\mathcal{R}$ un système de réécriture linéaire gauche et E un ensemble d'équations. Si la complétion termine sur $\mathcal{A}^*_{\mathcal{R},E}$, alors

$$\mathcal{L}(\mathcal{A}^*_{\mathcal{R},E}) \supseteq \mathcal{R}^*(\mathcal{L}(\mathcal{A})).$$

2.5 Timbuk

Pour expérimenter la complétion d'automates d'arbres, un outil est nécessaire. En effet, un automate d'arbres est une structure formelle difficile à représenter. De plus, le

Automate courant	$\mathcal{A}_0$	$\mathcal{A}^1_{\mathcal{R},E}$
Transitions ajoutées	$zero \to q_0$ $nil \to q_{nil}$ $cons(q_0, q_{nil}) \to q_{f_0}$	$s(q_0) \to q_1$ $s(q_1) \to q_2$ $cons(q_2, q_{f_0}) \to q_{f_1}$ $q_{f_1} \to q_{f_0}$
Paires critiques	$\langle cons(s(s(q_0)), cons(q_0, q_{nil}))\rangle$	$\langle cons(s(s(q_2)), cons(q_2, q_{f_0}))\rangle$
Application des équations ?	Non	Non
Langage	$\mathcal{L}(\mathcal{A}_0) = \{cons(zero, nil)\}$	$\mathcal{L}(\mathcal{A}^1_{\mathcal{R},E}) = \{cons(zero, nil)$ $cons(s(s(zero)), cons(zero, nil))\}$

Automate courant	$\mathcal{C}_{\mathcal{R}}(\mathcal{A}^1_{\mathcal{R},E})$	$\mathcal{A}^2_{\mathcal{R},E}$
Transitions ajoutées	$s(q_2) \to q_3$ $s(q_3) \to q_4$ $cons(q_4, q_{f_1}) \to q_{f_2}$ $q_{f_2} \to q_{f_1}$	$s(q_2) \to q_1$ $s(q_1) \to q_2$ $cons(q_2, q_{f_1}) \to q_{f_2}$ $q_{f_2} \to q_{f_1}$
Paires critiques	$\langle cons(s(s(q_4)), cons(q_4, q_{f_1}))\rangle$	-
Application des équations ?	$s(q_0) \to^* q_1$ et $s(s(s(q_0))) \to^* q_3$ $s(q_1) \to^* q_2$ et $s(s(s(q_1))) \to^* q_4$	$\Longrightarrow$ Fusion de q_1 et q_3 $\Longrightarrow$ Fusion de q_2 et q_4
Langage	$\mathcal{L}(\mathcal{C}_{\mathcal{R}}(\mathcal{A}^1_{\mathcal{R},E})) = \{cons(zero, nil)$ $cons(s(s(zero)), cons(zero, nil))$ $cons(s(s(s(s(zero)))),$ $cons(s(s(zero)), cons(zero, nil)))\}$	$\mathcal{L}(\mathcal{A}^2_{\mathcal{R},E}) = \{cons([s(s]^*(zero),$ $cons(\ldots, cons(zero, nil)\ldots)\}$

TABLEAU 2.1 — Étapes de complétion de l'exemple 2.24.

calcul de complétion n'est pas très adapté pour le raisonnement sur papier, étant donné la taille très importante des automates que ce calcul peut produire.

Pour cela, plusieurs outils ont été développés et rassemblés dans une même librairie appelée Timbuk [Genet et Viet Triem Tong, 2001b, Genet, 2008]. Nous allons ici décrire l'un d'entre eux qui permet d'effectuer la complétion d'automates d'arbres, puisque c'est celui qui sera utilisé et enrichi au cours de cette thèse. Cet outil est développé en Objective Caml [Leroy et al., 2000]. Son développement, débuté en 2000, continue encore et notamment dans cette thèse.

Timbuk implémente toutes les étapes de la complétion :

- l'algorithme de filtrage,
- la normalisation des nouvelles transitions,
- différentes abstractions telles que l'abstraction par un ensemble d'équations,
- le calcul de l'automate représentant l'ensemble des termes accessibles, ou une

sur-approximation de celui-ci,
- la décision du vide et la décision de l'inclusion, *i.e.* la vérification qu'aucun terme interdit n'est reconnu par l'automate calculé par complétion.

L'algorithme de complétion, et la vérification de non inclusion des termes interdits est effectuée à partir d'un fichier représentant la spécification du système, incluant le système de réécriture, l'automate d'arbres de départ (ou bien un ensemble fini de termes), l'ensemble des termes interdits (pouvant aussi être un automate d'arbres), et éventuellement une fonction d'abstraction.

Voici quelques exemples de spécification Timbuk et le résultat de leur exécution.

EXEMPLE *2.25*
Ce premier exemple décrit la concaténation de deux listes.

```
Ops append:2 nil:0 cons:2 a:0 b:0 c:0

Vars X Y P Q Z U

TRS R1
    append(nil,X) -> X
    append(cons(X,Y), Z) -> cons(X, append(Y,Z))

Automaton A0
    States qf qla qlb qnil qa qb
    Final States qf
    Transitions
         append(qla, qlb) -> qf
         cons(qa, qla) -> qla
         cons(qa, qnil) -> qla
         nil -> qnil
         cons(qb, qlb) -> qlb
         cons(qb, qnil) -> qlb
         a -> qa
         b -> qb

Patterns
    cons(a, cons(a, nil))
    cons(b, cons(a, _))
```

La structure de cette spécification est composée de :
- `Ops` : ensemble des symboles fonctionnels de l'alphabet ($\mathcal{F}$).
- `Vars` : ensemble des variables du système de réécriture ($\mathcal{X}$).
- `TRS R1` : `R1` est le système de réécriture représentant la concaténation de deux listes ($\mathcal{R}$).
- `Automaton A0` : automate d'arbres représentant les configurations initiales ($\mathcal{A}_0$), où :
  - `States` est l'ensemble des états ($\mathcal{Q}$),
  - `Final States` est l'ensemble des états finaux ($\mathcal{Q}_f$),
  - `Transitions` est l'ensemble des transitions (Δ).

Cet automate représente une liste non bornée de a et une liste non bornée de b qui vont être concaténées.

– `Pattern` : ensemble des termes interdits (Bad). Selon le premier terme, il ne doit pas y avoir de a à la fin de la liste. Les termes interdits peuvent être également décrits sous forme de *pattern*, *i.e.* possédant une certaine forme. Par exemple, le deuxième *pattern* permet de décrire qu'un b ne doit pas être suivi d'un a étant donné que la liste des a doit être en première position dans le résultat de la concaténation.

NB : L'ensemble des termes interdits peut également être représenté par un automate d'arbres, comme c'est le cas dans l'exemple 2.24.

L'exécution de Timbuk est réalisée grâce à l'édition de la commande suivante :

`timbuk 30 concatenation.txt,`

où `concatenation.txt` est le fichier à exécuter et `30` est un nombre choisi par l'utilisateur pour donner le nombre d'étapes maximum de complétion à effectuer avant d'arrêter le calcul.

Voici le résultat obtenu après exécution :

```
Completion step: 1
-------------------------------------------------------
New transitions: 3
New epsilon: 2

Completion step: 2
-------------------------------------------------------
New transitions: 0
New epsilon: 1

Completion step: 3
-------------------------------------------------------
New transitions: 0
New epsilon: 0

Export comp.res file...Done Export comp.res.epsi file...Done
Export comp.dot file...Done

States q0 q1 q2 q3 q4 q5 q6 q7 q8
Final States q0
Transitions
cons(q4,q0) -> q6
b -> q5
a -> q4
nil -> q3
cons(q5,q2) -> q2
cons(q5,q3) -> q2
cons(q4,q1) -> q1
cons(q4,q3) -> q1
cons(q4,q7) -> q8
```

```
append(q1,q2) -> q0
cons(q4,q0) -> q0
cons(q4,q7) -> q0
append(q3,q2) -> q7
cons(q5,q2) -> q7
cons(q5,q3) -> q7

Step: 3
Automaton complete!
------------------
Proof done!
-----------

Exported comp.res, comp.res.epsi and comp.dot files
Completion time: 0.001792 seconds
```

Le calcul s'arrête ici au bout de 3 étapes de complétion et l'automate final représentant l'ensemble des termes accessibles est donné. Le résultat `Proof done!` signifie qu'aucun terme interdit n'a été trouvé. Dans cet exemple, il n'y a pas de fonction d'approximation car la complétion s'exécute ici en un nombre fini d'étapes.

◀

EXEMPLE *2.26*
Ce deuxième exemple décrit l'algorithme du boulanger, qui peut être utilisé afin de réaliser une exclusion mutuelle sur toute machine multi-processeurs. L'algorithme reprend l'intuition de la gestion d'une file d'attente dans un petit commerce (boulangerie). Des numéros d'ordre croissant sont attribués par des tickets aux usagers au fur et à mesure qu'ils se présentent, et ces derniers sont servis dans l'ordre des numéros.

Les différents fils d'exécution (ou processus) souhaitant entrer en section critique sont donc les analogues des usagers. L'algorithme comporte schématiquement 3 phases:

- Attribution d'un numéro d'ordre (ticket).
- Attente de son tour avant l'entrée en section critique.
- Sortie de la section critique.

L'analogie ne peut cependant être poursuivie car, contrairement à ce qui se passe dans la vie courante, plusieurs fils d'exécution peuvent occasionnellement obtenir le même numéro d'ordre lors de la première phase, ce qui nécessite un arbitrage ultérieur. Cette implémentation de l'algorithme du boulanger représente deux processus voulant accéder à une même section critique. Chacun de ces processus possède un état qui est soit `sleep`, soit `wait` ou soit `crit`, ainsi qu'une valeur de ticket. Initialement, chaque processus est dans l'état `sleep` avec un ticket de valeur 0 (ici `o`). Quand un processus veut accéder à la section critique, il met sa valeur à la valeur du ticket de l'autre processus plus un et se met dans l'état `wait`. Un processus dans l'état `wait` se met dans l'état `crit` si la valeur du ticket de l'autre processus est 0 ou s'il a le plus petit ticket.

```
Ops state:4 o:0 s:1 sleep:0 wait:0 crit:0 true:0 false:0

Vars X Y P Q Z U
```

```
TRS R1
    state(sleep, X, Q, Y) -> state(wait, s(Y), Q, Y)
    state(wait, X, Q, o) -> state(crit, X, Q, o)
    state(wait, X, Q, s(Y)) -> state(crit, X, Q, Y) if X <-> Y
    state(crit, X, Q, Y) -> state(sleep, o, Q, Y)
    state(P, X, sleep, Y) -> state(P, X, wait, s(X))
    state(P, o, wait, Y) -> state(P, o, crit, Y)
    state(P, s(X), wait, Y) -> state(P, X, crit, Y) if X <-> Y
    state(P, X, crit, Y) -> state(P, X, sleep, o)

Set A1
    state(sleep,o,sleep,o)

Patterns
    state(crit,_,crit,_)

    Equations Abs
Rules
s(s(s(s(X))))=s(X)
```

Cette spécification est composée de :

- `TRS R1` : `R1` est le système de réécriture représentant l'algorithme du boulanger ($\mathcal{R}$).
- `Set A1` : ensemble des termes initiaux. Ici il s'agit d'un seul terme, et non pas un automate d'arbres.
- `Pattern` : ensemble des termes interdits (Bad). Deux processus ne peuvent pas être en section critique au même moment. Le *pattern* groupe ici tous les termes ayant le symbole `crit` en première et troisième position.
- `Equations` : l'ensemble des équations permettant de générer une sur-approximation calculable (E).

Pour exécuter ce fichier appelé `exempleBoulanger.txt` dans Timbuk, il suffit de taper la commande `timbuk 300 exempleBoulanger.txt`.

Voici un extrait du résultat obtenu après exécution :

```
Completion step: 1
-------------------------------------------------------
New transitions: 4
New epsilon: 2

[...]

Completion step: 12
-------------------------------------------------------
New transitions: 0
New epsilon: 0

Export comp.res file...Done
Export comp.res.epsi file...Done
Export comp.dot file...Done
```

```
States q0 q1 q2 q3 [...] q36 q37 q38
Final States q2
Transitions
state(q0,q1,q0,q1) -> q30
state(q0,q1,q3,q4) -> q30
state(q0,q1,q3,q10) -> q30
[...]
state(q7,q19,q3,q4) -> q32
state(q7,q19,q3,q19) -> q32
wait -> q3
crit -> q7
o -> q1
sleep -> q0

Step: 12
Automaton complete!
------------------
Proof done!
-----------

Exported comp.res, comp.res.epsi and comp.dot files
Completion time: 0.04508 seconds
```

Le calcul s'arrête au bout de 12 étapes de complétion et l'automate final représentant une sur-approximation de l'ensemble des termes accessibles est donné. Le résultat `Proof done!` est obtenu, signifiant qu'aucun terme interdit n'a été trouvé. Si le fichier est exécuté sans approximation (en ignorant la fonction d'approximation grâce à l'option `-noapprox`), alors les 300 étapes sont effectuées sans résultat (pas d'automate final), et il en est de même si on augmente le nombre d'étapes de complétion puisqu'il s'agit d'un système infini. ◀

Applications

Timbuk a été utilisé dans la vérification de protocoles de sécurité tels que le *Smartright system* [SmartRight, 2001] (voir [Genet, 2009]), ou encore pour vérifier des programmes $Java$ [Boichut et al., 2007] en utilisant l'outil Copster [Barré et al., 2009] que nous détaillerons dans le chapitre 6. Cet outil permet de transformer un fichier $Java$ `.class` en un système de réécriture (dans la spécification reconnue par Timbuk) représentant l'exécution du programme correspondant sur la JVM. En effet, les règles de réécriture permettent de représenter la sémantique du *bytecode* $Java$. Le fichier généré peut être agrémenté de termes interdits et d'une fonction d'abstraction pour ensuite être exécuté et vérifié par Timbuk. Nous verrons également dans le chapitre 6 une amélioration de la vérification des programmes $Java$ en utilisant Copster et Timbuk.

2.6 Treillis et interprétation abstraite

L'interprétation abstraite est une théorie générale permettant de trouver une approximation de la sémantique d'un programme. Cette théorie a été introduite par Cousot et Cousot dans [Cousot et Cousot, 1977]. L'abstraction de la sémantique du programme ainsi trouvée permet de faciliter la vérification et l'analyse statique du programme analysé, en inférant automatiquement des propriétés dynamiques des systèmes informatiques. Le premier principe de cette méthode est de définir une *sémantique concrète* C, représentant de façon précise le comportement du système, comme par exemple, dans le cas d'un programme impératif, toutes les traces d'exécutions possibles de ce programme.

EXEMPLE *2.27*
Soit le programme très simple suivant calculant la factorielle d'un nombre :

```
int fact (int n)
  int r = 1;
  for (int i=2; i<=n, i++) {
    r = r*i;
  }
  return r;
}
```

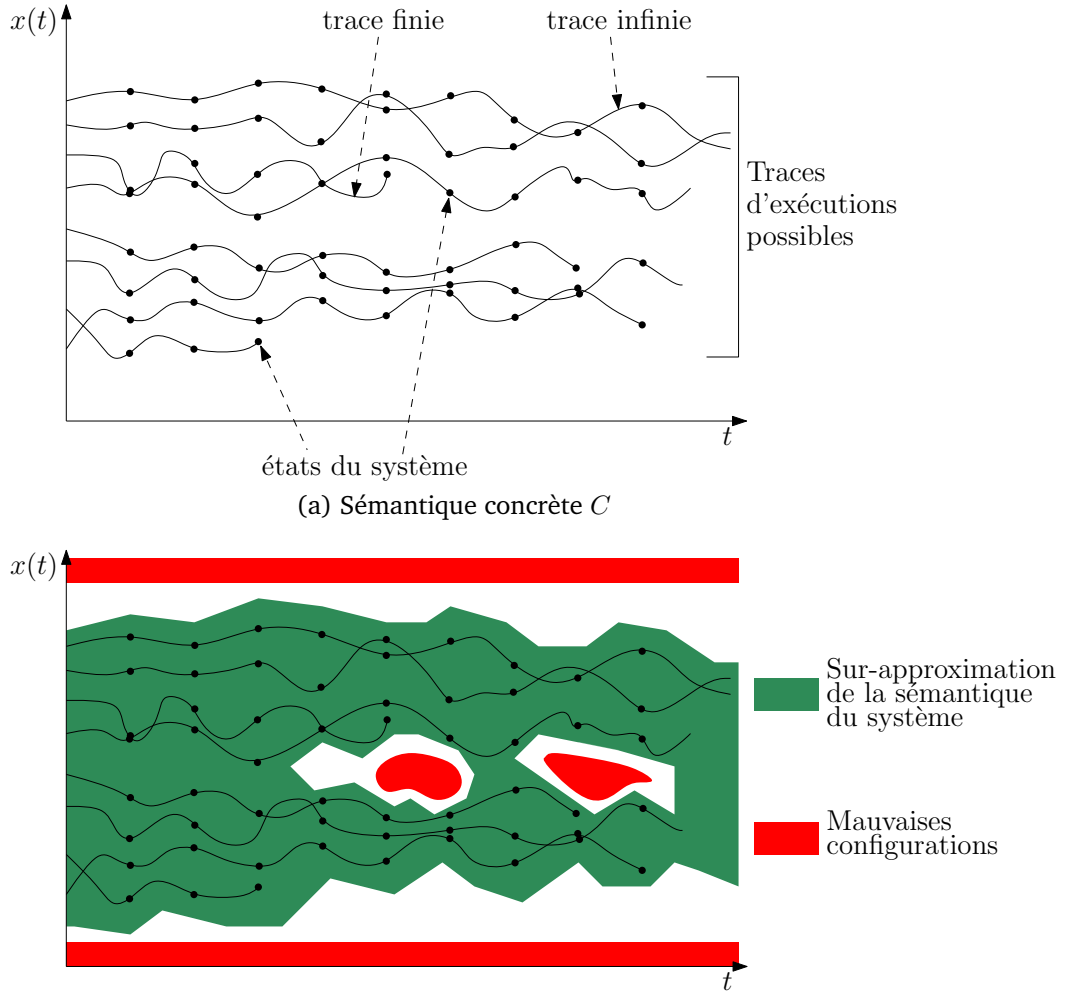
Alors voici la trace d'une possible exécution de ce programme pour `fact(4)` :
1 : `n` $\leftarrow 4$; `r` $\leftarrow 1$;
2 : `i` $\leftarrow 2$; `r` $\leftarrow 1 \times 2 = 1$;
3 : `i` $\leftarrow 3$; `r` $\leftarrow 2 \times 3 = 6$;
4 : `i` $\leftarrow 4$; `r` $\leftarrow 6 \times 4 = 24$;
5 : `i` $\leftarrow 5$;
6 : `return` 24; ◀

Mais cette sémantique concrète n'est en général pas calculable. Le second principe fondamental est de définir une abstraction A plus ou moins précise de la sémantique de C qui soit calculable. Comme il s'agit d'une approximation, elle est de ce fait souvent incomplète et doit être choisie soigneusement en fonction de la classe de propriétés à vérifier et du type de programme à analyser. Ainsi, sur la figure 2.14, la sémantique concrète (constituée de toutes les traces d'exécution du programme), est abstraite par une sur-approximation calculable, qui permet de vérifier que le programme n'atteint aucune mauvaise configuration.

L'abstraction du programme se fait généralement par une abstraction des domaines utilisés par le programme, comme l'ensemble des entiers relatifs ou des réels, à l'aide un treillis complet, que nous allons définir dans ce qui suit.

2.6.1 Treillis et leurs propriétés

Le postulat de base de l'interprétation abstraite est que toute sémantique peut être exprimée comme un ensemble de valeurs prises dans un domaine C. Le domaine concret C peut ensuite être abstrait par un treillis complet.



(a) Sémantique concrète C

(b) Sémantique abstraite A

FIGURE 2.14 — Abstraction de la sémantique d'un programme.

DÉFINITION 2.6.1 (Ensemble partiellement ordonné (*poset*))
Un ensemble partiellement ordonné (ou poset) est un doublet $(\Lambda, \sqsubseteq)$ *avec* Λ *un ensemble et* $\sqsubseteq$ *une relation d'ordre partiel, i.e. respectant les conditions suivantes :*

- *Réflexivité :* $\forall x \in \Lambda,\ x \sqsubseteq x$
- *Antisymétrie :* $\forall x, y \in \Lambda,\ x \sqsubseteq y \ \wedge\ y \sqsubseteq x \ \Rightarrow\ x = y$
- *Transitivité :* $\forall x, y, z \in \Lambda,\ x \sqsubseteq y \ \wedge\ y \sqsubseteq z \ \Rightarrow\ x \sqsubseteq z$

DÉFINITION 2.6.2 (Treillis)
Un treillis est un poset $(\Lambda, \sqsubseteq)$ *où chaque couple d'élément de* Λ *admet une plus petite borne supérieure* $\sqcup$ *(least upper bound : lub) et une plus grande borne inférieure* $\sqcap$ *(greatest lower bound : glb), i.e. :*

- $\forall x, y \in \Lambda,\ x \sqsubseteq x \sqcup y \ \wedge\ y \sqsubseteq x \sqcup y$
- $\forall x, y, z \in \Lambda,\ x \sqsubseteq z \ \wedge\ y \sqsubseteq z \ \Rightarrow\ x \sqcup y \sqsubseteq z$
- $\forall x, y \in \Lambda,\ x \sqcap y \sqsubseteq x \ \wedge\ x \sqcap y \sqsubseteq y$
- $\forall x, y, z \in \Lambda,\ z \sqsubseteq x \ \wedge\ z \sqsubseteq y \ \Rightarrow\ z \sqsubseteq x \sqcap y$

Un treillis est complet si les opérateurs $\sqcap$ (*glb*) et $\sqcup$ (*lub*) sont définis pour tout sous-

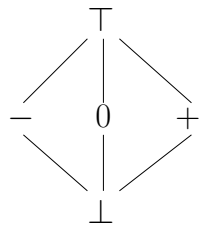


FIGURE 2.15 — Le treillis des signes.

ensemble possiblement infini de Λ. Un treillis complet possède nécessairement un plus grand élément $\top$, et un plus petit élément $\bot$.

EXEMPLE *2.28*
Le treillis des signes $S = \{\top, \bot, 0, +, -\}$ représenté en figure 2.15 est complet, et on a par exemple $+ \sqcup - = \top$ (*lub*), $0 \sqcap + = \bot$ (*glb*). ◀

DÉFINITION 2.6.3 (Treillis atomique)
Un élément x d'un treillis complet $(\Lambda, \sqsubseteq)$ est un atome s'il est minimal, i.e. $\bot \sqsubset x \wedge \forall y \in \Lambda : \bot \sqsubset y \sqsubseteq x \Rightarrow y = x$. Un ensemble d'atomes de Λ est noté $Atoms(\Lambda)$. Un treillis complet $(\Lambda, \sqsubseteq)$ est atomique si tout élément $x \in \Lambda$ avec $x \neq \bot$ est une borne supérieure d'atomes, i.e. :

$$x = \sqcup\{a | a \in Atoms(\Lambda) \wedge a \sqsubseteq x\}.$$

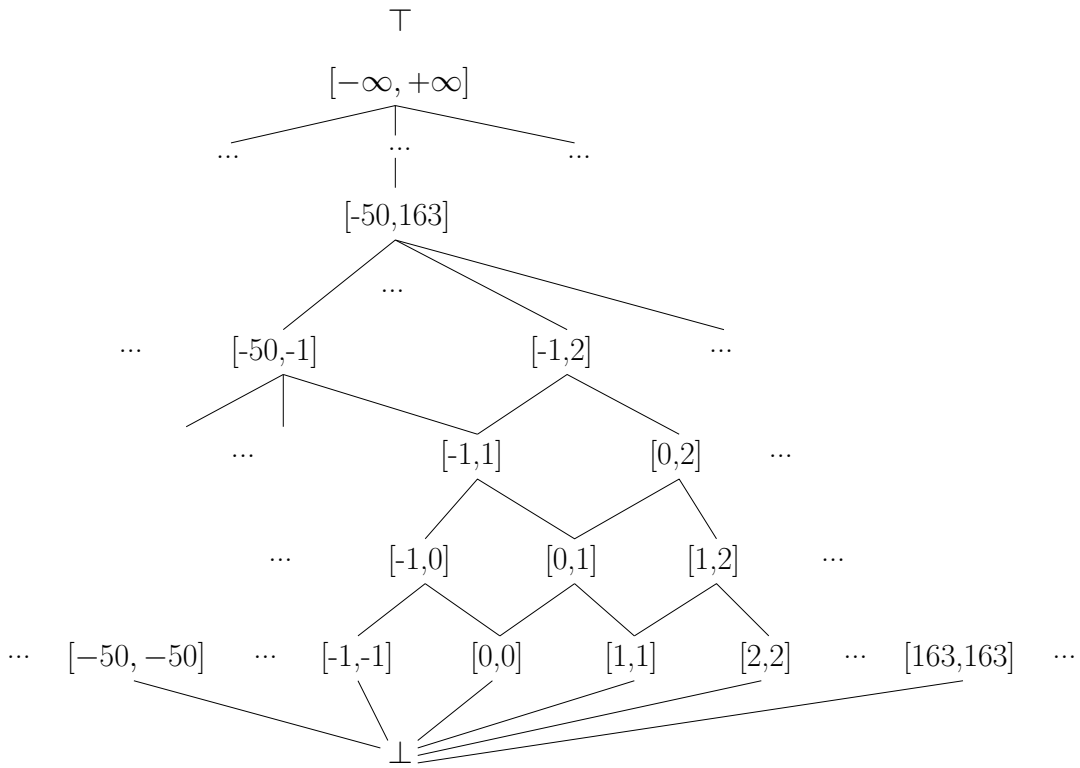


FIGURE 2.16 — Le treillis des intervalles est atomique.

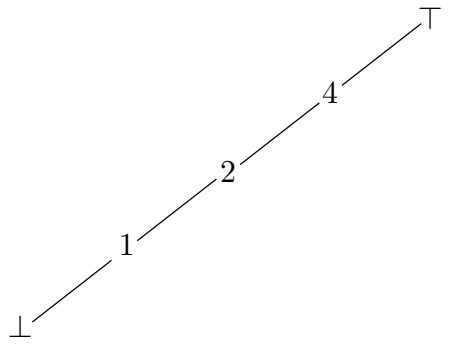


FIGURE 2.17 — Le treillis des diviseurs de 4 n'est pas atomique.

EXEMPLE *2.29*
Les ensembles d'entiers $(2^{\mathbb{Z}}, \subseteq)$ peuvent être abstraits par le treillis $(\Lambda, \sqsubseteq)$ des intervalles, dont les bornes appartiennent à $\mathbb{Z} \cup \{-\infty, +\infty\}$. Ce treillis est représenté en figure 2.16. Il s'agit d'un treillis atomique et ses atomes sont de la forme $[x, x]$, pour chaque $x \in \mathbb{Z}$.

En revanche, le treillis des diviseurs de 4 (représenté en figure 2.17) avec l'ordre "est diviseur de" n'est pas atomique. En effet, l'élément 4 ne peut pas être obtenu en effectuant une borne supérieure d'atomes, car le seul atome de ce treillis est l'élément 1. ◀

Les treillis généralement utilisés en interprétation abstraite sont :
- les intervalles (ex : $x \in [4, 10]$),
- les octogones (ex : $x - y < 3$),
- les polyèdres (ex : $2x + 4y \geq 10$),
- les ellipsoïdes (ex : $(x - 3)^2 + (y - 10)^2 \leq 34$),
- etc.

Sur la figure 2.18 sont représentés tous les états d'un programme issus de ces traces d'exécutions, ainsi que les différentes façons de les abstraire par des treillis.

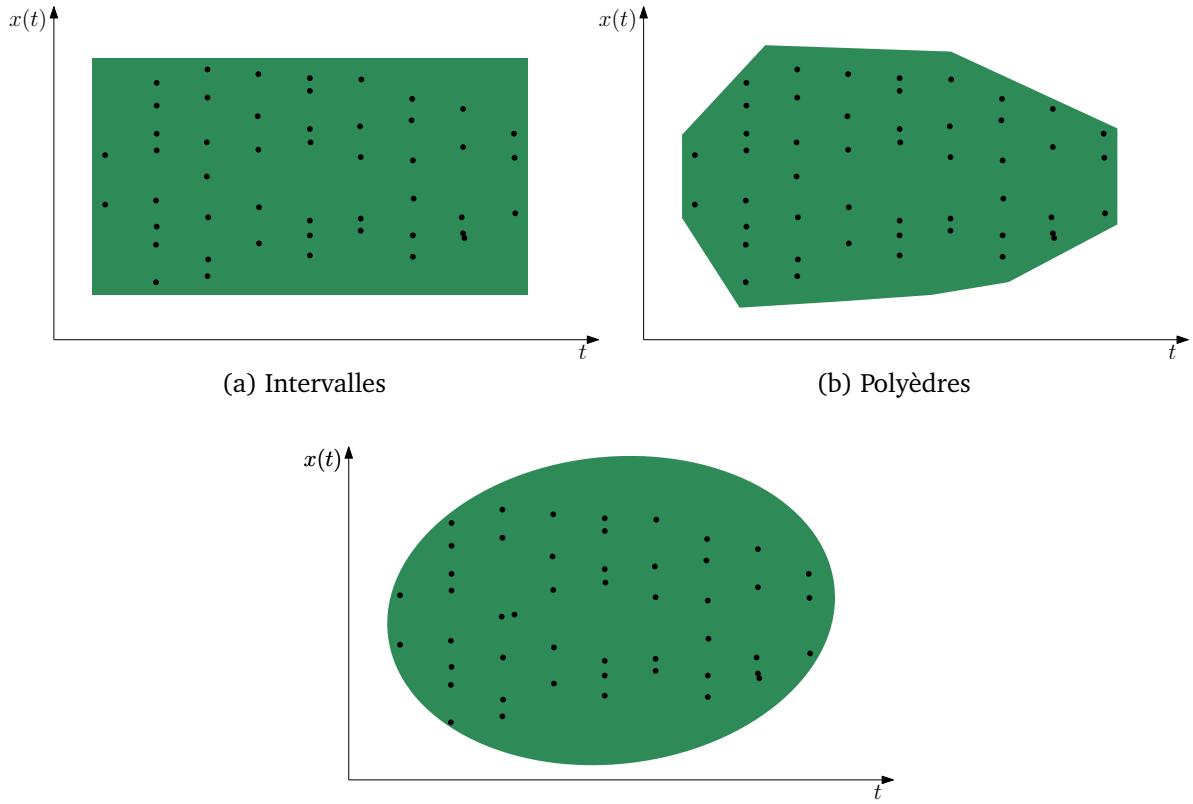
2.6.2 Connexions de Gallois

La sémantique concrète doit être remplacée par un domaine abstrait afin de générer une analyse calculable. Pour que le domaine abstrait choisi soit correct, il doit être dérivé de la sémantique concrète, et pour cela respecter ce que l'on appelle les connexions de Gallois.

DÉFINITION 2.6.4 (Connexions de Gallois)
Si on considère deux treillis $(C, \sqsubseteq_C)$ (le domaine concret) et $(A, \sqsubseteq_A)$ (le domaine abstrait), alors on dit qu'il y a une connexion de Galois *entre ces deux treillis s'il y a deux fonctions monotones $\alpha : C \to A$ (fonction d'abstraction) et $\gamma : A \to C$ (fonction de concrétisation) telles que :*

$$\forall x \in C, y \in A,\ \alpha(x) \sqsubseteq_A y \text{ si et seulement si } x \sqsubseteq_C \gamma(y).$$

Si α et γ sont données pour une abstraction A de C, alors A est une abstraction correcte de C, et ceci permet de garantir que A abstrait correctement les informations de C.



(a) Intervalles (b) Polyèdres

(c) Ellipsoïdes

FIGURE 2.18 — Différents types de treillis pour l'abstraction.

EXEMPLE *2.30*
Soit C l'ensemble des parties de $\mathbb{Z}$ ($\mathcal{P}(\mathbb{Z})$), et A le treillis des signes $\{\top, \bot, 0, +, -\}$ de l'exemple 2.28 et de la figure 2.15. Alors on peut définir une connexion de Gallois entre A et C grâce aux fonctions α_{sign} et γ_{sign} définies comme suit. Pour chaque sous ensemble e de C on a :

- $\alpha_{sign}(e) = +$ si tous les éléments de e sont plus grand que zéro,
- $\alpha_{sign}(e) = -$ si tous les éléments de e sont plus petit que zéro,
- $\alpha_{sign}(\{0\}) = 0$,
- $\alpha_{sign}(\emptyset) = \bot$, et
- $\alpha_{sign}(e) = \top$ sinon.
- $\gamma_{sign}(+) = \mathbb{Z}_+$,
- $\gamma_{sign}(-) = \mathbb{Z}_-$,
- $\gamma_{sign}(0) = \{0\}$,
- $\gamma_{sign}(\bot) = \emptyset$, et
- $\gamma_{sign}(\top) = \mathbb{Z}$,

où $\mathbb{Z}_+ = \{x \in \mathbb{Z} \mid x > 0\}$, $\mathbb{Z}_- = \{x \in \mathbb{Z} \mid x < 0\}$. ◀

2.6.3 Opérateur de *widening*

Nous avons vu dans la section 2.4 que pour terminer le calcul de complétion, il fallait obtenir un automate point-fixe $\mathcal{A}^i_{\mathcal{R}}$, *i.e.* tel que $\mathcal{A}^i_{\mathcal{R}} = \mathcal{C}(\mathcal{A}^i_{\mathcal{R}})$.

Dans la plupart des problèmes d'analyse de programmes, il s'agit également de résoudre une équation point-fixe $x = F(x)$ où x appartient à un ensemble ordonné C. Nous avons vu ci-dessus comment abstraire ce domaine par un domaine abstrait A. Malgré cette abstraction, la résolution itérative de l'équation point-fixe peut quand même entrainer des itérations infinies. On utilise alors ce qu'on appelle un opérateur de *widening* à partir d'un nombre d'itérations p donné. Un opérateur de widening ∇ permet d'abstraire une séquence infinie en devinant le comportement futur de la séquence. La séquence originale $y_0 = init$, $y_{n+1} = F(y_n)$ est remplacée par $z_0 = init$, $z_p = z_{p-1} \nabla F(z_{p-1})$. Dans le cas de la complétion d'automates d'arbres, $init$ serait l'automate de départ $\mathcal{A}_0$ et $F(x)$ une étape de complétion appliquée sur l'automate x soit $\mathcal{C}(x)$. Sous réserve de quelques hypothèses techniques sur ∇, la convergence du calcul itératif de la séquence $(z_n)_{n \geq 0}$ est garantie.

EXEMPLE *2.31*
Reprenons le treillis des intervalles décrit dans l'exemple 2.29. Soit $f(x) \rightarrow f(x+1)$ une règle de réécriture. L'équation point-fixe correspondante serait alors $f(x) = f(x+1)$, et la séquence infinie générée depuis le terme $f([0,1])$ serait la suivante : $y_0 = f([0,1]), y_1 = f([1,2]), y_2 = f([2,3]), \ldots, y_{123} = f([123,124]), \ldots$ Dans le cadre de l'interprétation abstraite, l'opérateur de *widening* ∇ ne peut pas être utilisé directement sur un terme (tel que $f([1,2])$, mais il est appliqué seulement sur les intervalles.

Supposons qu'on puisse raisonner sur des termes comprenant des intervalles. Ce que l'on souhaite obtenir revient à appliquer le widening sur la partie intervalle du terme. Alors, si l'opérateur de *widening* ∇ est utilisé à partir de la quatrième étape d'itération, on obtient la séquence suivante : $z_0 = [0,1], z_1 = [1,2], z_2 = [2,3], z_3 = z_2 \nabla (z_2 + 1) = [2, +\infty[$. On obtient alors le terme abstrait $f([2, +\infty[)$. De la même manière, si la séquence infinie est décroissante, alors elle sera abstraite par $]-\infty, x]$ pour $x \in \mathbb{Z}$. L'application de l'opérateur de *widening* sur les termes sera résolue au chapitre 5. ◀

État de l'art et comparaisons

3

Sommaire

3.1 Introduction **56**

3.2 Model-Checking régulier et transducteurs **57**

3.2.1 Model-Checking régulier avec automates de mots : *Regular Model-Checking* 57

3.2.2 Model-Checking régulier avec automate d'arbres : *Regular Tree Model-Checking* 62

3.3 Fonctions d'abstractions du model-checking régulier **69**

3.3.1 Problématique et exemple fil rouge 69

3.3.2 Fusion d'états par équivalence de langage 71

3.3.3 Fusion d'états par équivalence de prédicats 72

3.3.4 Autres fonctions d'abstractions de l'outil Timbuk 74

3.3.5 Autres techniques d'abstractions basées sur le *widening* 75

3.3.6 Raffinement automatique depuis un faux contre-exemple . . . 76

3.4 Model-Checking avec `Maude` **79**

3.4.1 L'outil `Maude` 79

3.4.2 Équations - Modules fonctionnels 81

3.4.3 Règles de réécritures - Modules système 82

3.4.4 Model-Checking 84

3.4.5 Comparaison `Maude`/Timbuk 90

3.5 Application concrète à l'analyse de programme **92**

3.5.1 Traduction exacte de la sémantique 92

3.5.2 Construction d'un modèle abstrait du programme 94

3.6 Conclusion et ouverture sur les contributions **99**

"Ils ne savaient pas que c'était impossible, alors ils l'ont fait."

Mark Twain

3.1 Introduction

Dans ce chapitre, nous allons placer la méthode utilisée dans cette thèse par rapport aux autres méthodes similaires de vérification formelles. En effet, cette thèse utilise et améliore la complétion d'automates d'arbres, qui possède de nombreux avantages. Elle permet notamment la vérification de certains systèmes intéressants tels que les protocoles de sécurité ou les programmes $Java$. Rappelons brièvement, tel qu'expliqué dans le chapitre 2, que cette méthode utilise les automates d'arbres afin de représenter des ensembles de configurations du système que l'on souhaite vérifier, et qu'elle emploie un système de réécriture pour modéliser la fonction de transition de ce système. Nous avons également établi que le calcul des configurations accessibles retenu dans le cadre de cette thèse est effectué grâce à un algorithme de *complétion* (voir section 2.4). Mais bien d'autres techniques de vérification formelle existent.

Notons que la modélisation retenue pour représenter un système est une modélisation régulière : en effet, nous avons choisi d'utiliser des automates d'arbres. Cela présente certains avantages : les automates de mots ou d'arbres sont des modélisations efficaces, pour lesquelles les opérations nécessaires à la vérification d'un système (telle que l'intersection, l'union, le complément, l'inclusion) sont décidables (voir chapitre 2).

Dans ce chapitre, nous allons donc comparer la méthode que nous avons retenue à quelques techniques similaires de vérification. La première section de ce chapitre détaille d'autres méthodes utilisant la représentation régulière par automates. Nous les comparerons ensuite avec la méthode choisie pour permettre d'expliquer notre choix. La première méthode décrite, appelée le *Regular Model-Checking* [Bouajjani et al., 2000], représente une configuration du système est représentée par un mot, et un ensemble potentiellement infini de configurations par un automate de mots. La fonction de transition, permettant de représenter la comportement du système à vérifier, est représentée par un transducteur. La seconde méthode décrite, appelée le *Regular Tree Model-Checking* [Abdulla et al., 2002], utilise les automates d'arbres de la même manière que la complétion, mais le comportement du système est modélisé par un transducteur d'arbres plutôt que par un système de réécriture.

Nous avons vu dans la section 2.4 que la vérification de systèmes à états infinis nécessite des fonctions d'abstraction afin de calculer une sur-approximation des configurations accessibles du système. Nous avons alors détaillé la méthode d'abstraction équationnelle, mais d'autres méthodes existent dans le cadre du Model-Checking régulier, et nous allons les décrire en deuxième section de ce chapitre.

Ensuite, nous comparerons l'implémentation `Timbuk` avec l'outil `Maude` [Clavel et al., 2009], qui permet, par une modélisation utilisant des équations et des règles de réécriture, de réécrire et réduire des termes typés. Cet outil permet entre autres de vérifier des propriétés temporelles.

Enfin, nous situerons la technique choisie dans le domaine de l'analyse de programmes d'un point de vu applicatif, pour la vérification de programmes concrets en $Java$, ou en C, ou encore certains protocoles de sécurité.

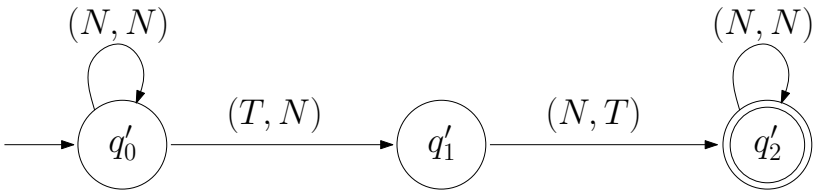


FIGURE 3.1 — Transducteur modélisant le passage de jeton entre deux processus.

3.2 *Regular (Tree) Model-Checking* : calcul de l'ensemble des configurations accessibles par transducteurs

3.2.1 Model-Checking régulier avec automates de mots : *Regular Model-Checking*

Principe général

Cette méthode, décrite dans [Bouajjani et al., 2000], se situe dans le cadre d'un système état-transitions comme expliqué en section 2.3. Ainsi, un mot représente une configuration du système, un automate de mots représente un ensemble (possiblement infini) de configurations du système, et un (ou une composition de) transducteur(s) de mots représente le comportement du système, *i.e.* ses transitions.

Rappelons que les définitions concernant les automates de mots sont disponibles en section 2.1. Commençons tout d'abord par définir ce qu'est un transducteur de mots.

DÉFINITION 3.2.1 (Transducteur de mots)
Soit Σ et Σ' deux alphabets finis. Un transducteur fini de mots est un tuple $\tau = \langle \Sigma \times \Sigma', \mathcal{Q}, \mathcal{Q}_0, \mathcal{Q}_f, \delta \rangle$, avec $\mathcal{Q}$ un ensemble fini d'états, $\mathcal{Q}_0 \subseteq \mathcal{Q}$ l'ensemble des états initiaux, $\mathcal{Q}_f \subseteq \mathcal{Q}$ l'ensemble d'états finaux et $\delta \subseteq \mathcal{Q} \times \Sigma \times \Sigma' \times \mathcal{Q}$ un ensemble de transitions.

La relation de transition $\rightarrow \subseteq \mathcal{Q} \times \Sigma^* \times \Sigma^* \times \mathcal{Q}$ est définie comme étant la plus petite relation satisfaisant :

(1) $\forall q \in \mathcal{Q} : q \xrightarrow{\epsilon,\epsilon} q$,

(2) si $(q, a, b, q') \in \delta$, *alors* $q \xrightarrow{(a,b)} q'$ *et*

(3) si $q \xrightarrow{(a,c)} q'$ *et* $q' \xrightarrow{(b,d)} q''$, *alors* $q \xrightarrow{(ab,cd)} q''$

Un transducteur peut être vu comme un automate composé d'entrées/sorties. Il lit un mot en entrée, et en produit un autre en sortie. Le langage (ou la relation) reconnu(e) par un transducteur correspond à l'ensemble des paires de mots (w_1, w_2) telles que le transducteur lisant w_1 en entrée produit w_2 en sortie.

EXEMPLE *3.1*
Nous pouvons voir sur la figure 3.1 un exemple de transducteur, modélisant le passage du jeton entre deux processus adjacents (voir exemple 2.2). En effet si un jeton T appartient à un processus i, alors il lui est enlevé (ceci est modélisé par la transition (q_0, T, N, q_1)), et le processus suivant prend le jeton (ceci est modélisé par la transition (q_1, N, T, q_2)). Soit "$NNTNN$" le mot représentant un ensemble de 5 processus, dont le

3e possède le jeton T. L'application de ce transducteur sur "$NNTNN$" renvoie le mot "$NNNTN$". C'est donc désormais le 4e processus qui possède le jeton T, il y a bien eu passage du jeton entre deux processus successifs. ◀

Nous avons vu en section 2.3 le principe du Model-Checking régulier. Rappelons que dans le cadre de la vérification de propriétés de sûreté, ce Model-Checking permet de calculer l'ensemble des configurations accessibles par le système, afin de vérifier qu'il n'admet aucune configuration interdite. Ainsi dans l'algorithme de complétion d'automates d'arbres décrit en section 2.4, le but est de calculer l'ensemble $\mathcal{R}^*(\mathcal{L}(\mathcal{A}_0))$ ou une sur-approximation de cet ensemble, pour un automate d'arbres initial $\mathcal{A}_0$ donné. Soit $\mathcal{A}_0$ l'automate de mots initial et τ le transducteur de mots représentant les transitions du système, nous voulons alors calculer l'ensemble $\tau^*(\mathcal{L}(\mathcal{A}_0))$, τ^* étant la *clôture réflexive et transitive* de τ.

DÉFINITION 3.2.2 (Clôture réflexive et transitive de τ (τ^*))
Soit τ un transducteur, on note τ^i le transducteur obtenu par composition de τ i fois avec lui-même, et $\tau^0 = \tau_{id}$. Alors $\tau^* = \bigcup_{i=0}^{\infty} \tau^i$.

L'ensemble $\tau^*(\mathcal{L}(\mathcal{A}_0))$, ou une sur-approximation de cet ensemble, se calcule par l'application successive du transducteur de mots sur l'automate de départ, comme pour la complétion d'automates d'arbres :

$$\mathcal{A}_0 \xrightarrow{\tau} \mathcal{A}_1 \xrightarrow{\tau} \mathcal{A}_2 \xrightarrow{\tau} \ldots \xrightarrow{\tau} \mathcal{A}_f,$$

avec $\mathcal{L}(\mathcal{A}_1 \cup \mathcal{A}_2 \cup \ldots \cup \mathcal{A}_f) \supseteq \tau^*(\mathcal{L}(\mathcal{A}_0))$. Rappelons que l'union d'automates peut être trouvée dans [Comon et al., 2007]. Nous remarquons également que le successeur $\mathcal{A}_{i+1}$ est calculé par application de τ sur $\mathcal{A}_i$, mais ne reconnait pas le langage de $\mathcal{A}_i$, ce qui diffère du calcul de complétion. En effet, $\mathcal{A}_{i+1}$ reconnait seulement le langage successeur calculé.

L'application d'un transducteur de mots sur un automate $\mathcal{A}_i$ dans le but de calculer l'automate successeur $\mathcal{A}_{i+1} = \tau(\mathcal{A}_i)$ (ou encore l'image de $\mathcal{A}_i$ par τ) est définie de la manière suivante.

DÉFINITION 3.2.3 (Calcul de $\tau(\mathcal{A})$)
Soit $\mathcal{A}$ un automate de mots et $\tau = \langle \Sigma \times \Sigma', \mathcal{Q}, \mathcal{Q}_0, \mathcal{Q}_f, \delta \rangle$ un transducteur. La projection $\Pi^{\neq 1}$ sur un transducteur, est la projection sur le deuxième élément des transitions du transducteur, soit $\Pi^{\neq 1}(\tau) = \langle \Sigma', \mathcal{Q}, \mathcal{Q}_0, \mathcal{Q}_f, \delta_{\Pi^{\neq 1}} \rangle$, où $\delta_{\Pi^{\neq 1}} = \{(q, b, q') \mid (q, a, b, q') \in \delta\}$. L'image de $\mathcal{A}$ par τ est donnée par :

$$\tau(\mathcal{A}) = \Pi^{\neq 1}[(\mathcal{A} \times \mathcal{A}^{\Sigma}) \cap \tau],$$

où $\mathcal{A}^{\Sigma}$ est l'automate minimal de Σ^*, soit un automate fini acceptant tous les mots possibles construits sur Σ, soit Σ^*.

Remarque. Un transducteur diffère d'un automate par la présence d'une paire sur chaque transition plutôt qu'un singleton. Notons donc que dans la définition précédente, $\mathcal{A}^{\Sigma}$ permet de transformer $\mathcal{A}$ en transducteur, *i.e.* possédant des paires (a, b) sur chaque transition. Le produit de $\mathcal{A}$ et $\mathcal{A}^{\Sigma}$ retourne donc un transducteur dont le premier élément a de chaque paire (a, b) est le mot présent sur la transition correspondante de $\mathcal{A}$, et

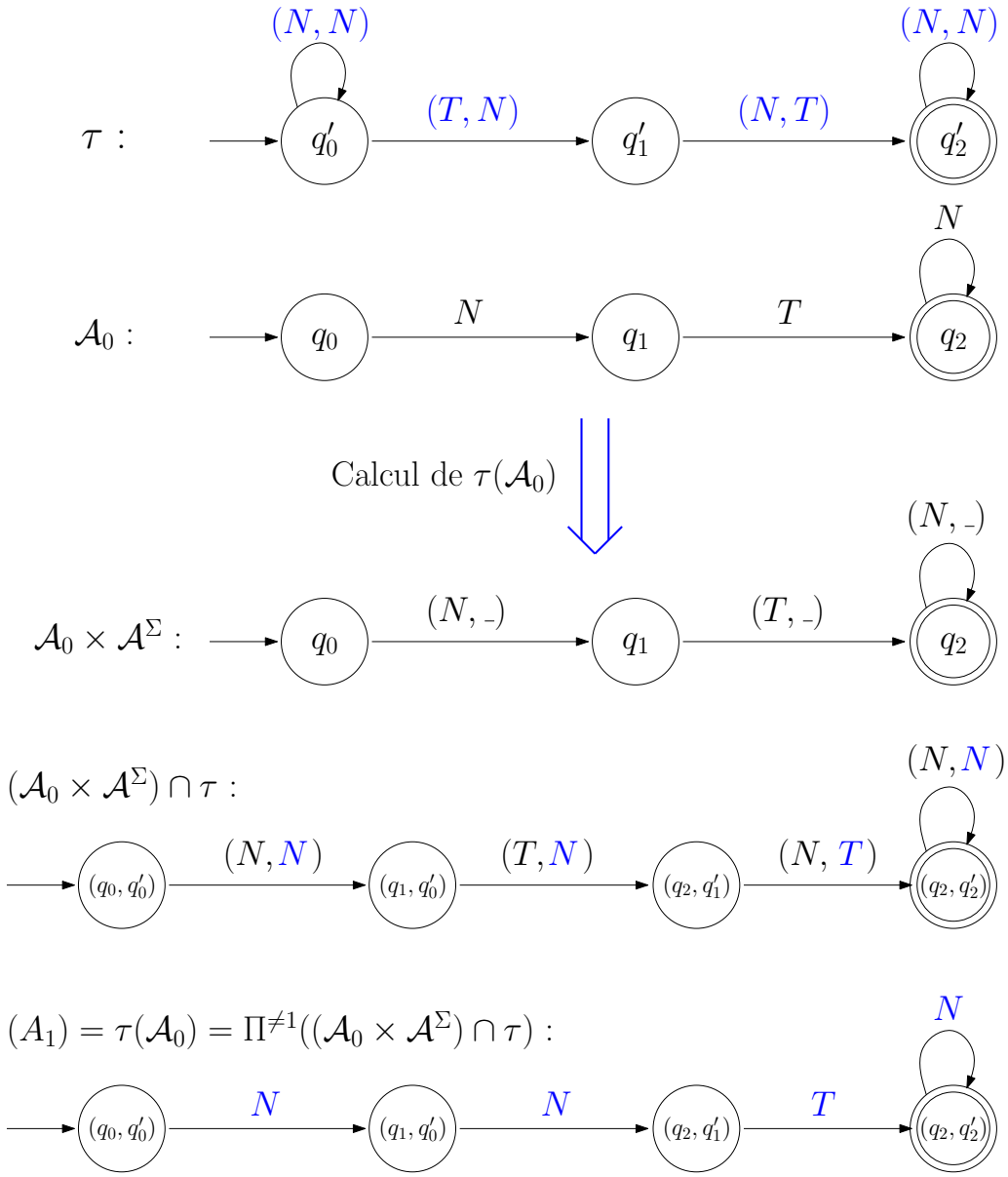


FIGURE 3.2 — Calcul de $\tau(\mathcal{A}_0)$ pour le protocole du passage de jeton.

dont le deuxième élément b contient n'importe quel symbole de Σ. Ce produit est donc effectué pour pouvoir calculer l'intersection entre $\mathcal{A}$ et τ. Rappelons que la définition de l'intersection entre deux automates est disponible dans [Comon et al., 2007].

EXEMPLE *3.2*
La figure 3.2 présente l'application du transducteur τ de la figure 3.1 sur l'automate $\mathcal{A}_0$ de la figure 2.1. Ainsi l'automate $\mathcal{A}_1$ résultat représente l'ensemble des configurations pour lesquelles le 3e processus possède le jeton. ◀

Soient $\mathcal{A}_0$ l'automate représentant l'ensemble des configurations initiales du système à vérifier, τ le transducteur représentant son comportement, et Bad l'ensemble de mots interdits. Il faut alors calculer l'ensemble des configurations accessibles $\tau^*(\mathcal{L}(\mathcal{A}_0))$. Rappelons que cet ensemble est l'union des langages des automates successeurs calculés

par application successive du transducteur (voir définition 3.2.3 et exemple 3.2). Il suffit ensuite de vérifier que $\tau^*(\mathcal{L}(\mathcal{A}_0)) \cap Bad = \emptyset$. Cependant, dans le cas de systèmes à états infinis, l'ensemble $\tau^*(\mathcal{L}(\mathcal{A}_0))$ n'est pas toujours calculable. Dans certains cas, la clôture τ^* est calculable, mais le transducteur τ doit être réflexif, transitif, et respecter d'autres critères détaillés dans [Boigelot et al., 2003]. Dans le cas où le calcul de τ^* et $\tau^*(\mathcal{L}(\mathcal{A}_0))$ n'est pas terminant, des fonctions d'abstractions permettant de calculer une sur-approximation de $\tau^*(\mathcal{L}(\mathcal{A}_0))$ ont été définies dans [Bouajjani et al., 2004], et seront évoquées en section 3.3.

Comparaison automate de mots/automate d'arbres

Les automates de mots finis, tout comme les automates d'arbres finis, sont des structures très utilisées en informatique théorique, notamment en vérification, et leurs opérations (intersection, union, complément, etc.) sont décidables et s'effectuent en un temps polynomial. Les automates de mots sont plus simples à utiliser, manipuler et représenter graphiquement, tandis qu'il n'existe pas à l'heure actuelle de représentation graphique simple et intuitive des automates d'arbres. Une des seules manières de représenter un automate d'arbres utilise des hypergraphes orientés comme détaillé dans [Bouajjani et Touili, 2002].

DÉFINITION 3.2.4 (Hypergraphe et automate d'arbres)
Soit S un ensemble de sommets et $\mathcal{F}$ un alphabet de symboles fonctionnels. Soit $f \in \mathcal{F}^n$ et $s, s_1, \ldots, s_n \in S$, alors $s \xrightarrow{f} s_1, \ldots, s_n$ est une hyper-arête orientée. Un hypergraphe orienté est une paire (S, H) où S est un ensemble de sommets et H un ensemble d'hyper-arêtes orientées sur S.

Soient $\mathcal{A} = \langle \mathcal{F}, \mathcal{Q}, \mathcal{Q}_f, \Delta \rangle$ un automate d'arbres, l'ensemble de transitions Δ peut être représenté par un hypergraphe orienté $(\mathcal{Q}, H_\Delta)$, où H_Δ est défini par :
- $q \xrightarrow{a} \in H_\Delta$ *pour toute transition* $a \to q \in \Delta$,
- $q \xrightarrow{f} q_1, \ldots, q_n \in H_\Delta$ *pour toute transition* $f(q_1, \ldots, q_n) \to q \in \Delta$,
- $q \to q' \in H_\Delta$ *pour toute transition* $q \to q' \in \Delta$

EXEMPLE *3.3*
La figure 3.3 donne une représentation en hypergraphe orienté de l'automate d'arbres contenant l'ensemble de transitions suivant :

$$\begin{aligned}\Delta = \{ \quad & a \to q_a, b \to q_b \\ & i(q_f, q_j) \to q_i, j(q_j, q_a) \to q_j, \\ & k(q_b, q_f) \to q_k, f(q_i, q_j, q_k) \to q_f \ \}\end{aligned}$$

Comme nous pouvons le voir sur cette figure, il est nécessaire d'utiliser des arcs spéciaux permettant de modéliser les différents fils d'un état, et il n'y a pas d'indication sur l'ordre des fils, à part la lecture de gauche à droite. Cette représentation peut toutefois être utilisée pour détecter des répétitions entre deux automates d'arbres successeurs lors d'un calcul des accessibles [Bouajjani et Touili, 2002], comme nous le verrons en section 3.3.5. ◀

Cependant, un automate d'arbres permet une plus grande expressivité. En effet, certains systèmes sont beaucoup mieux représentés par des fonctions.

Imaginons que l'on veuille représenter une fonction par un mot. Alors celui-ci doit être bien parenthésé, *i.e.* contenir autant de parenthèses ouvrantes que de parenthèses

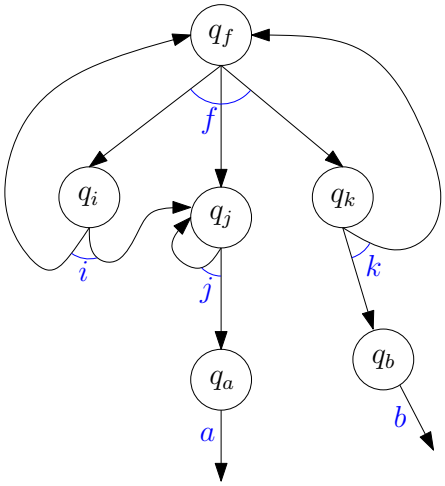


FIGURE 3.3 — Représentation d'un automate d'arbres par un hypergraphe orienté.

fermantes, comme on le voit par exemple sur cette fonction : $f(g(x, h(h(y))))$. Or dans le cas des mots, un langage contenant des expressions bien parenthésée avec un nombre non borné de parenthèses ne peut pas être un langage régulier, puisqu'il faudrait alors compter le nombre de parenthèses ouvrantes et fermantes. Il est donc impossible de représenter un tel langage à l'aide d'un automate de mots fini : il faudrait par exemple utiliser un automate à pile.

EXEMPLE *3.4*
Le langage d'arbres régulier $\mathcal{L}_A = f^*(g(a, h^*(b)))$, équivaut dans le cas des mots au langage $\mathcal{L}_M = [f(]^n(g(a,[h(]^m b)^m)^n$, ce qui n'est pas un langage régulier, puisque le nombre de parenthèses ouvrantes et fermantes doit être compté. ◀

Contrairement aux mots, les arbres permettent effectivement de représenter parfaitement la hiérarchie entre les différents éléments d'un système. C'est le cas notamment :

- des structures des phrases dans le traitement automatique des langues naturelles [Joshi et al., 1975]. En effet, un arbre va permettre de catégoriser et hiérarchiser les différentes parties d'une phrase, comme nous pouvons le voir par exemple sur la figure 3.4.
- de la structure d'un document `XML`. En effet la structure en balises hiérarchisées est typiquement une structure en arbre.
- des états d'un bytecode d'un programme $Java$ [Boichut et al., 2007]. En effet, un état d'un programme $Java$ est composé d'une frame d'exécution courante f, d'une pile de frame fs, d'un tas h pour stocker les différents objets du programme, et d'un tas statique k permettant de stocker les valeurs des champs statiques, et ainsi un état se modélise parfaitement par un terme $s(f, fs, h, k)$. De la même manière, une frame est composée de différents éléments et se représente également de manière intuitive par un arbre $f(m, pc, s, l)$. Plus de détails concernant la modélisation des programmes $Java$ se trouvent en section 6.2.
- du fonctionnement d'un protocole de sécurité. En effet, les messages envoyés durant le protocole sont facilement représentable par des termes de la forme $msg(x, y, c)$, où x et y sont respectivement les agents émetteurs et récepteurs du message, et où c est le contenu du message. Les agents se représentent également

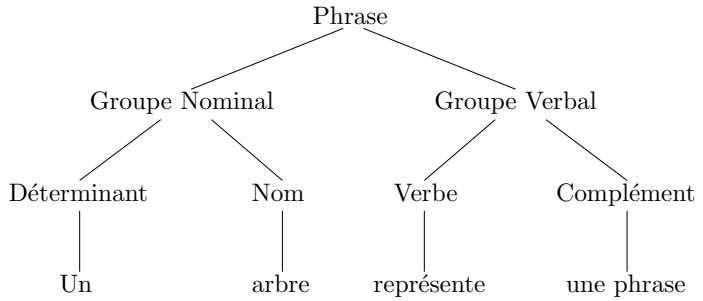


FIGURE 3.4 — Un arbre permet de structurer les éléments d'une phrase.

par un terme $agt(l)$ où l est l'étiquette de l'agent, le contenu d'un message peut être la clé publique d'un agent a ($pubkey(a)$), ou encore le résultat du cryptage d'un contenu c par une clé k, qui se représente par le terme $encr(k, c)$. Plus de détails pourront être trouvés dans [Genet et Klay, 2000].

3.2.2 Model-Checking régulier avec automate d'arbres : *Regular Tree Model-Checking*

Principe général

Dans le chapitre 2, nous avons introduit la complétion d'automates d'arbres, permettant la vérification d'un système par l'utilisation d'automates d'arbres et d'un système de réécriture. Nous avons également mentionné que notre objectif consiste à calculer l'ensemble $\mathcal{R}^*(\mathcal{L}(\mathcal{A}))$ (ou une sur-approximation) étant donné un automate $\mathcal{A}$ initial. Cet ensemble est obtenu par applications successives de $\mathcal{R}$. Une fois cet ensemble calculé, il suffit de vérifier qu'il ne contient aucune configuration interdite, et si c'est le cas, la propriété est vérifiée.

Suivant le même principe, et utilisant également les automates d'arbres pour modéliser un ensemble de configurations, les auteurs de [Abdulla et al., 2002] utilisent non pas un système de réécriture mais un transducteur (ou une composition de transducteurs) d'arbres τ pour modéliser le comportement du système à vérifier. Donnons désormais la définition générale d'un transducteur d'arbres, disponible dans [Comon et al., 2007, Bouajjani et Touili, 2002].

DÉFINITION 3.2.5 (Transducteur d'arbres)
Soit $\mathcal{F}$ et $\mathcal{F}'$ deux alphabets finis. Un transducteur fini d'arbres bottom-up (soit transducteur d'arbres, pour plus de concision) est un tuple $\tau = \langle \mathcal{F}, \mathcal{F}', \mathcal{Q}, \mathcal{Q}_f, \mathcal{X}, \Delta \rangle$, avec $\mathcal{Q}$ un ensemble fini d'états, $\mathcal{Q}_f \subseteq \mathcal{Q}$ un ensemble d'états finaux, $\mathcal{X}$ un ensemble de variables et Δ un ensemble de transitions de la forme :

- $a \to q(u)$, *avec* $u \in \mathcal{T}(\mathcal{F}')$, $a \in \mathcal{F}^0$ *et* $q \in \mathcal{Q}$, *ou*
- $q(x) \to q'(u)$, *avec* $u \in \mathcal{T}(\mathcal{F}', \{x\})$, $q, q' \in \mathcal{Q}$ *et* $x \in \mathcal{X}$, *ou*
- $f(q_1(x_1), \ldots, q_n(x_n)) \to q(u)$, *avec* $u \in \mathcal{T}(\mathcal{F}', \{x_1, \ldots, x_n\})$, $f \in \mathcal{F}^n$, $q, q_1, \ldots, q_n \in \mathcal{Q}$ *et* $x_1, \ldots, x_n \in \mathcal{X}$.

La clôture réflexive et transitive τ^* d'un transducteurs d'arbres τ est la même que dans le cas des transducteurs de mots (voir définition 3.2.2).

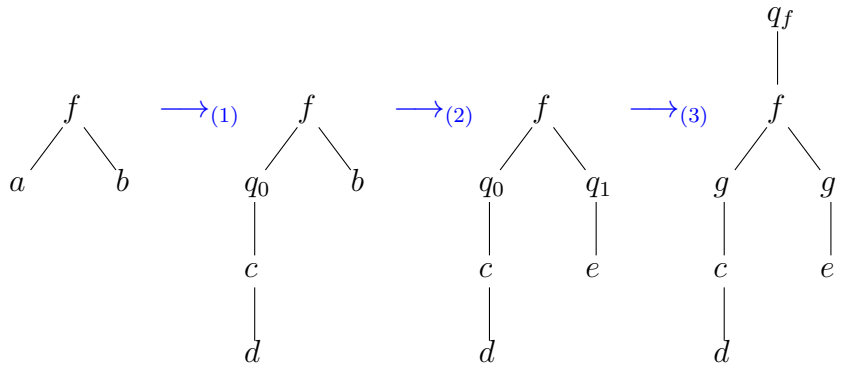


FIGURE 3.5 — Exécution du transducteur τ de l'exemple 3.5 sur le terme $f(a, b)$.

EXEMPLE *3.5*
Soit $\tau = \langle \mathcal{F}, \mathcal{F}', \mathcal{Q}, \mathcal{Q}_f, \mathcal{X}, \Delta \rangle$, avec $\mathcal{F} = \{f_2, a_0, b_0\}$, $\mathcal{F}' = \{f_2, c_1, g_1, d_0, e_0\}$, $\mathcal{Q} = \{q_0, q_f\}$, $\mathcal{Q}_f = \{q_f\}$, et

$$\Delta = \{ \quad a \to q_0(c(d)) \ \ (1) \ , b \to q_1(e) \ \ (2) \ , \\ f(q_0(x), q_1(y)) \to q_f(f(g(x), g(y))) \ \ (3) \ \}.$$

L'exécution d'un transducteur d'arbres sur un terme fonctionne quasiment de la même manière que l'exécution d'un automate d'arbres sur terme, à la différence qu'un symbole n'est pas remplacé seulement par un état mais par un état contenant un nouveau terme. Ainsi, l'exécution du transducteurs τ sur le terme $f(a, b)$ est détaillé en figure 3.5, où les numéros (en bleu) correspondent à la transition de Δ utilisée. ◀

Pour un système de réécriture $\mathcal{R}$ et un automate $\mathcal{A}^i_{\mathcal{R}}$, le calcul de l'automate successeur $\mathcal{A}^{i+1}_{\mathcal{R}}$ est effectué grâce à l'algorithme de complétion décrit en section 2.4. Dans le calcul des configurations accessibles grâce à un transducteur d'arbres τ, le calcul de l'automate successeur correspond au calcul de l'automate $\tau(\mathcal{A}_i)$. Ce calcul du successeur peut être différencié selon la catégorie de τ : *structure-preserving* ou **non** *structure-preserving*. Un transducteur d'arbres *structure-preserving* conserve la structure du terme passé en entrée, *i.e.* consiste simplement en un réétiquetage des nœuds du terme, tandis qu'un transducteur **non** *structure-preserving* modifie la structure du terme passé en entrée.

EXEMPLE *3.6*
Le transducteur défini dans l'exemple 3.5 est clairement **non** *structure-preserving*. En effet, la figure 3.5 montre que la structure du terme $f(a, b)$ de départ est modifiée. Soit τ_{st} un transducteur dont l'ensemble de transition Δ_{st} est le suivant :

$$\Delta_{st} = \{a \to q_0(b), f(q_0(x), q_0(y)) \to q_f(g(x, y))\}.$$

Alors τ_{st} est *structure-preserving* : en effet, il se contente de réétiqueter le symbole a en b et le symbole f en g, sans modifier la structure du terme. Ainsi, le terme $f(a, a)$ est transformé en $g(b, b)$ qui possède la même structure. ◀

Cette distinction a été créée car le calcul de $\tau(\mathcal{A})$ dans le cas *structure-preserving* est beaucoup plus simple que dans le cas **non** *structure-preserving*. Les automates *structure-preserving* sont ainsi plus utilisés dans le domaine du *Regular Tree Model-Checking*, no-

tamment dans certains calculs des configurations accessibles basés sur l'itérations de transducteurs [Abdulla et al., 2005], seulement définis dans le cas *structure-preserving*.

Calcul de $\tau(\mathcal{A})$ dans le cas *structure-preserving*

Introduisons tout d'abord des définitions différentes des automates et des transducteurs d'arbres, afin de donner une définition formelle du calcul de $\tau(\mathcal{A})$ dans le cas *structure-preserving*. Ces définitions sont disponibles dans [Abdulla et al., 2005].

DÉFINITION 3.2.6 (Automate d'arbres - formalisme différent)
Soit $\mathcal{A} = \langle \mathcal{F}, \mathcal{Q}, \mathcal{Q}_f, \Delta \rangle$ un automate d'arbres. Alors les transitions de Δ peuvent également être de cette forme :

- $\xrightarrow{a} q$, *avec* $a \in \mathcal{F}^0$ *et* $q \in \mathcal{Q}$,
- $(q_1, \ldots, q_n) \xrightarrow{f} q$, *avec* $f \in \mathcal{F}^n$ *et* $q, q_1, \ldots, q_n \in \mathcal{Q}$,
- $q \xrightarrow{\epsilon} q'$, *avec* $q, q' \in \mathcal{Q}$ *(ϵ-transition).*

DÉFINITION 3.2.7 (Transducteur d'arbres *structure-preserving*)
Soit $\tau = \langle \mathcal{F}, \mathcal{F}', \mathcal{Q}, \mathcal{Q}_f, \mathcal{X}, \Delta \rangle$. Alors τ est structure-preserving si ses transitions peuvent s'écrire sous la forme suivante :

- $\xrightarrow{a,b} q$, *avec* $a \in \mathcal{F}^0, b \in (\mathcal{F}')^0$ *et* $q \in \mathcal{Q}$,
- $(q_1, \ldots, q_n) \xrightarrow{f,g} q$, *avec* $f \in \mathcal{F}^n, g \in (\mathcal{F}')^n$ *et* $q, q_1, \ldots, q_n \in \mathcal{Q}$,

L'ensemble de variables $\mathcal{X}$ n'est alors plus nécessaire dans le cas structure-preserving.

Dans le cas *structure-preserving*, le calcul de $\tau(\mathcal{A})$ pour un automate $\mathcal{A}$ donné est très similaire au cas des mots. En effet, l'application d'un transducteur de mots sur un automate $\mathcal{A}_i$ dans le but de calculer l'automate successeur $\mathcal{A}_{i+1} = \tau(\mathcal{A}_i)$ (ou encore l'image de $\mathcal{A}_i$ par τ) est définie de la manière suivante.

DÉFINITION 3.2.8 (Calcul de $\tau(\mathcal{A})$ dans le cas *structure-preserving*)
Soit $\mathcal{A}$ un automate d'arbres et $\tau = \langle \mathcal{F}, \mathcal{F}', \mathcal{Q}, \mathcal{Q}_f, \Delta \rangle$ un transducteur d'arbres structure-preserving.

La projection $\Pi^{\neq 1}$ sur un transducteur d'arbres, est la projection sur le deuxième élément des transitions du transducteur, soit $\Pi^{\neq 1}(\tau) = \langle \mathcal{F}', \mathcal{Q}, \mathcal{Q}_f, \Delta_{\Pi^{\neq 1}} \rangle$, où $\Delta_{\Pi^{\neq 1}} = \{(q_1, \ldots, q_n) \xrightarrow{g} q \mid (q_1, \ldots, q_n) \xrightarrow{f,g} q \in \Delta\}$.

L'image de $\mathcal{A}$ par τ est donnée par :

$$\tau(\mathcal{A}) = \Pi^{\neq 1}[(A \times \mathcal{A}^{\mathcal{F}}) \cap \tau],$$

où $\mathcal{A}^{\mathcal{F}}$ est un automate d'arbres acceptant tous les termes possibles construits sur $\mathcal{F}$.

EXEMPLE *3.7*
Soit $\mathcal{A}$ un automate d'arbres et τ un transducteur d'arbres *structure-preserving* dont les transitions sont énumérées dans le tableau 3.1. Alors le calcul de $\tau(\mathcal{A})$ est résumé dans le tableau 3.1. ◀

$\mathcal{A}$:	τ :
$\{ \xrightarrow{a} q_0,$ $\xrightarrow{b} q_1,$ $(q_0, q_1) \xrightarrow{f} q_2,$ $q_2 \xrightarrow{h} q_3 \}$	$\{ \xrightarrow{(a,c)} q'_0,$ $\xrightarrow{(b,d)} q'_1,$ $(q'_0, q'_1) \xrightarrow{(f,g)} q'_2,$ $(q'_1, q'_0) \xrightarrow{(g,f)} q'_3 \}$
$\mathcal{A}\times\mathcal{A}^{\mathcal{F}}$:	$\tau \cap A \times\mathcal{A}^{\mathcal{F}}$:
$\{ \xrightarrow{(a,_)} q_0,$ $\xrightarrow{(b,_)} q_1,$ $(q_0, q_1) \xrightarrow{(f,_)} q_2,$ $q_2 \xrightarrow{(h,_)} q_3 \}$	$\{ \xrightarrow{(a,c)} (q_0, q'_0),$ $\xrightarrow{(b,d)} (q_1, q'_1),$ $((q_0, q'_0), (q_1, q'_1)) \xrightarrow{(f,g)} (q_2, q'_2) \}$

$\tau(A) = \Pi^{\neq 1}(\tau \cap A \times\mathcal{A}^{\mathcal{F}}) =$
$\{ \xrightarrow{c} (q_0, q'_0)$ $\xrightarrow{d} (q_1, q'_1)$ $((q_0, q'_0), (q_1, q'_1)) \xrightarrow{g} (q_2, q'_2) \}$

TABLEAU 3.1 — Calcul de $\tau(\mathcal{A})$ dans le cas *structure-preserving* (exemple 3.7).

Calcul de $\tau(\mathcal{A})$ dans le cas non *structure-preserving*

Nous allons ici définir le calcul de $\tau(\mathcal{A})$ dans le cas **non** *structure-preserving* grâce à l'algorithme suivant. Cet algorithme utilise la définition 3.2.5 des transducteurs d'arbres.

DÉFINITION 3.2.9 (Calcul de $\tau(\mathcal{A})$ dans le cas **non** *structure-preserving*)
Soit $\mathcal{A} = \langle \mathcal{F}, \mathcal{Q}, \mathcal{Q}_f, \Delta \rangle$ un automate d'arbres et $\tau = \langle \mathcal{F}, \mathcal{F}', \mathcal{Q}', \mathcal{Q}'_f, \mathcal{X}, \Delta' \rangle$ un transducteur d'arbres. Alors l'automate d'arbres obtenu par application de τ sur $\mathcal{A}$ est l'automate $\tau(\mathcal{A}) = \langle \mathcal{F}', \mathcal{Q} \times \mathcal{Q}', \mathcal{Q}_f \times \mathcal{Q}'_f, \Delta_{\tau(\mathcal{A})} \rangle$, où $\Delta_{\tau(\mathcal{A})}$ est calculé grâce à l'algorithme suivant :

Initialisation : $\Delta_{\tau(\mathcal{A})} := \emptyset$.

Étape 1 :

(1) Pour tout $a \to q \in \Delta$ et $a \to q'(u) \in \Delta'$, avec $a \in \mathcal{F}^0$ et $u \in \mathcal{T}(\mathcal{F}')$,
$\Delta_{\tau(\mathcal{A})} := \Delta_{\tau(\mathcal{A})} \cup \{Norm(u \to (q, q'))\}$.

(2) Pour tout $f(q_1, \ldots, q_n) \to q \in \Delta$ et $f(q'_1(x_1), \ldots, q'_n(x'_n)) \to q'(u[x_{i_1}, \ldots, x_{i_k}]) \in \Delta'$, avec $f \in \mathcal{F}^n$, $u[x_{i_1}, \ldots, x_{i_k}] \in \mathcal{T}(\mathcal{F}', \{x_{i_1}, \ldots, x_{i_k}\})$,
$\Delta_{\tau(\mathcal{A})} := \Delta_{\tau(\mathcal{A})} \cup \{Norm(u[(q_{i_1}, q'_{i_1}), \ldots, (q_{i_k}, q'_{i_k})] \to (q, q'))\}$,
tel que $\forall j \in [1, k], \exists i \in [1, n]$ tel que $x_{i_j} = x_i$, $q_{i_j} = q_i$ et $q'_{i_j} = q'_i$.

Étape 2 : Tant que l'on peut ajouter des transitions à $\Delta_{\tau(\mathcal{A})}$, faire

(3) Pour tout $q_1 \to q_2 \in \Delta$ et $v \to (q_1, q'_1) \in \Delta_{\tau(\mathcal{A})}$,
$\Delta_{\tau(\mathcal{A})} := \Delta_{\tau(\mathcal{A})} \cup \{(q_1, q'_1) \to (q_2, q'_1)\}$.

(4) Pour tout $q'_1(x) \to q'_2(u(x)) \in \Delta'$ et $v \to (q_1, q'_1) \in \Delta_{\tau(\mathcal{A})}$, avec $u \in \mathcal{T}(\mathcal{F}', \{x\})$,
$\Delta_{\tau(\mathcal{A})} := \Delta_{\tau(\mathcal{A})} \cup \{Norm(u((q_1, q'_1)) \to (q_1, q'_2))\}$.

(5) Pour tout $q'_1(x) \to q'_2(u) \in \Delta'$ *et* $v \to (q_1, q'_1) \in \Delta_{\tau(\mathcal{A})}$*, avec* $u \in \mathcal{T}(\mathcal{F}')$*,*
$\Delta_{\tau(\mathcal{A})} := \Delta_{\tau(\mathcal{A})} \cup \{Norm(u \to (q_1, q'_2))\}$.

<u>NB</u> : $Norm(u \to q)$ *est la fonction de normalisation de la définition 2.4.6, permettant de transformer la transition* $u \to q$ *potentiellement non normalisée en plusieurs transitions normalisées (voir définition 2.2.2).*

Notons que l'algorithme tel que nous l'avons défini ci-dessus est une adaptation d'une définition incomplète de la composition d'un transducteur d'arbres et d'un automate d'arbres qui peut être trouvée dans [Touili, 2003].

EXEMPLE *3.8*
Soit $\mathcal{A}$ un automate d'arbres possédant un ensemble Δ de transitions et τ un transducteur d'arbres possédant un ensemble Δ' de transitions. Δ et Δ' sont alors les ensembles suivants :

$\mathcal{A}$			τ		
$\Delta = \{$	$a \to q_0,$	(A)	$\Delta' = \{$	$a \to q'_1(b),$	(A')
	$b \to q_2,$	(B)		$b \to q'_2(a),$	(B')
	$f(q_1, q_2) \to q_f,$	(C)		$q'_2(x) \to q'_3(h(h(x))),$	(C')
	$q_f \to q_{f_{bis}}$ $\}$	(D)		$f(q'_1(x), q'_2(y)) \to q'_f(f(g(x), g(x))),$	(D')
				$g(q'_1(x)) \to q'_4(h(h(x)))$	(E') $\}$

On applique désormais l'algorithme de la définition 3.2.9 afin de calculer l'ensemble $\Delta_{\tau(\mathcal{A})}$ des transitions de l'automate $\tau(\mathcal{A})$, obtenu par application de τ sur $\mathcal{A}$.
Étape 1 : $\Delta_{\tau(\mathcal{A})} = \emptyset$

- Dans un premier temps, nous allons utiliser la règle (1) de l'algorithme 3.2.9 sur les transitions (A) et (A'). En effet, on a $a \to q_0 \in \Delta$ et $a \to q'_1(b) \in \Delta'$, donc $\Delta_{\tau(\mathcal{A})} := \{Norm(b \to (q_1, q'_1))\}$.
- Puis nous allons utiliser encore une fois la règle (1) avec les transitions (B) et (B'), ce qui va donner $\Delta_{\tau(\mathcal{A})} := \Delta_{\tau(\mathcal{A})} \cup \{Norm(a \to (q_2, q'_2))\}$.
- Règle (2) avec transitions (C) et (D') :
  $\Delta_{\tau(\mathcal{A})} := \Delta_{\tau(\mathcal{A})} \cup \{Norm(f(g((q_1, q'_1)), g((q_1, q'_1))) \to (q_f, q'_f))\}$.

Soit q_g un nouvel état, *i.e.* n'appartenant ni à $\mathcal{A}$, ni à τ. Après normalisation des transitions, on alors pour l'instant :

$\Delta_{\tau(\mathcal{A})} = \{$	$b \to (q_1, q'_1),$	($A_{\Delta_{\tau(\mathcal{A})}}$)	
	$a \to (q_2, q'_2),$	($B_{\Delta_{\tau(\mathcal{A})}}$)	
	$g((q_1, q'_1)) \to q_g,$	($C_{\Delta_{\tau(\mathcal{A})}}$)	
	$f(q_g, q_g) \to (q_f, q'_f)$	($D_{\Delta_{\tau(\mathcal{A})}}$)	$\}$

Étape 2 :

- Règle (3) avec transitions (D) de Δ et ($D_{\Delta_{\tau(\mathcal{A})}}$) du $\Delta_{\tau(\mathcal{A})}$ courant :
  $\Delta_{\tau(\mathcal{A})} = \Delta_{\tau(\mathcal{A})} \cup \{(q_f, q'_f) \to (q_{f_{bis}}, q'_f)\}$.
- Règle (4) avec transitions (C') et ($B_{\Delta_{\tau(\mathcal{A})}}$) :
  $\Delta_{\tau(\mathcal{A})} = \Delta_{\tau(\mathcal{A})} \cup \{Norm(h(h((q_2, q'_2))) \to (q_2, q'_3))\}$.

La transition (E') de τ ne peut pas être appliquée ici car il n'y a pas de transitions commençant par le symbole g dans $\mathcal{A}$. Soit q_h un nouvel état, alors l'automate successeur, *i.e.* $\tau(\mathcal{A})$ possède donc l'ensemble de transitions suivant.

$\tau(\mathcal{A})$
$\Delta_{\tau(\mathcal{A})} = \{ \quad b \to (q_1, q'_1),$
$a \to (q_2, q'_2),$
$g((q_1, q'_1)) \to q_g,$
$f(q_g, q_g) \to (q_f, q'_f),$
$(q_f, q'_f) \to (q_{f_{bis}}, q'_f),$
$h((q_2, q'_2)) \to q_h,$
$h(q_h) \to (q_2, q'_3) \quad \}$

◀

Comme nous l'avons vu dans le cas des mots, dans le cas de systèmes à états infinis, l'ensemble $\tau^*(\mathcal{L}(\mathcal{A}_0))$ pour un automate initial $\mathcal{A}_0$ donné n'est pas toujours calculable. La clôture τ^* peut être calculable, mais seulement dans le cas *structure-preserving* [Abdulla et al., 2005]. Dans le cas **non** *structure-preserving*, des fonctions d'abstractions permettant de calculer une sur-approximation de $\tau^*(\mathcal{L}(\mathcal{A}_0))$ ont été définies dans [Bouajjani et al., 2006a], et seront évoquées en section 3.3.

Comparaison système de réécriture/transducteur d'arbres.

Les systèmes de réécriture et les transducteurs d'arbres sont relativement différents, ce qui rend la comparaison difficile. Cependant, nous pouvons dire que les systèmes de réécriture sont Turing-complets et peuvent donc représenter le comportement de n'importe quel système, tandis que les transducteurs d'arbres ont une expressivité limitée. Ceci vient, entre autre, du fait que les transitions d'un transducteurs d'arbres doivent être normalisées, *i.e.* ne peuvent pas contenir plus d'un symbole dans la partie gauche : la transition $f(q(x)) \to q_f(f(g(x)))$ est autorisée, mais on ne peut pas avoir la transition $f(g(q(x))) \to q_f(g(f(x)))$. Cependant, les transducteurs d'arbres sont Turing-complets par *composition*, *i.e.* un seul transducteur ne peut pas tout représenter, mais par une composition de plusieurs transducteurs cela devient possible. Mais cette composition peut rendre la modélisation plus complexe et moins intuitive qu'avec système de réécriture, comme nous allons le voir dans l'exemple suivant.

EXEMPLE *3.9*
La concaténation de deux listes peut être représentée par le système de réécriture suivant, contenant seulement deux règles :

$$\mathcal{R} = \{append(cons(x, y), z) \to cons(x, append(y, z)), append(nil, x) \to x\}.$$

Or un transducteur seul ne permet pas de représenter la concaténation de deux listes. Prenons la première règle $append(cons(x, y), z) \to cons(x, append(y, z))$. Pour pouvoir la représenter par des transducteurs d'arbres, il faut alors effectuer la composition des deux transducteurs τ_1 et τ_2 suivants :

τ_1 :	τ_2 :
$\{ \quad a \to q(a),$	$\{ \quad cons(q(x), q(y)) \to q_{c1}(x),$
$nil \to q(nil),$	$cons(q(x), q(y)) \to q_{c2}(y),$
$cons(q(x), q(y)) \to q_c(cons(x, y)),$	$1st(q_{c1}(x)) \to q_1(x),$
$append(q_c(u), q(z))$	$2nd(q_{c2}(x)) \to q_2(x),$
$\to q(cons(1st(u)), append(2nd(u), z)) \quad \}$	$append(q_2(x), q(y)) \to q_f(x),$
	$cons(q_1(x), q_f(y)) \to q_f(cons(x, y)) \quad \}$

◀

Par ailleurs, le calcul des configurations accessibles, par l'intermédiaire des calculs successifs de $\tau(\mathcal{A})$, est assez complexe dans le cas **non** *structure-preserving*, et la nécessité d'une éventuelle composition de transducteurs rajoute une complexité supplémentaire dans le calcul de l'automate successeur. La plupart des travaux appliquant le *Regular Tree Model-Checking* utilisent donc des transducteurs *structure-preserving* (comme c'est le cas dans [Abdulla et al., 2005]), et concernent donc un ensemble limité de systèmes. En effet, il est intéressant de noter que les transducteurs *structure-preserving* correspondent à une classe de réécriture nommée L-SM, pour laquelle le calcul des termes accessibles est *décidable* [Genet, 2009]. Toute transition $(q_1, \ldots, q_n) \xrightarrow{f,g} q$ d'un transducteur *structure-preserving* peut se représenter par une règle de réécriture $f(x_1, \ldots, x_n) \to g(x_1, \ldots, x_n)$.

Cette méthode de calcul des accessibles via des transducteurs d'arbres (incluant les fonctions d'abstractions que nous allons voir en section 3.3), a été implémentée dans un prototype [Bouajjani et al., 2006a]. Ce prototype utilise les opérations basiques nécessaires sur les automates d'arbres (telles que l'intersection, l'union, l'inclusion, ...) fournies par la librairie de Timbuk [Genet, 2008]. Timbuk a donc été étendu en ajoutant une implémentation permettant de gérer les transducteurs d'arbres et le calcul de $\tau(\mathcal{A})$. Il y a dans ce prototype deux implémentations différentes de ce calcul : une assez simple pour les transducteurs *structure-preserving*, et une plus complexe pour les **non** *structure-preserving*, nécessitant une décomposition du transducteur en trois parties plus simples telle qu'elle est expliquée dans les travaux [Engelfriet, 1975].

Toutefois, la restriction d'expressivité d'un transducteur d'arbres lui confère une propriété intéressante que les systèmes de réécriture n'ont pas. Comme nous l'avons vu dans cette section, il est possible de construire un automate d'arbres $\mathcal{A}_{i+1}$ tel que $\mathcal{L}(\mathcal{A}_{i+1}) = \tau(\mathcal{L}(\mathcal{A}_i))$. Ceci peut être vu comme *une seule étape* d'application du transducteur τ, *i.e.* une seule étape *dans le temps*, ce qui rend possible la vérification de *propriétés temporelles*. Soient $\mathcal{R}$ un système de réécriture, $\mathcal{T}$ un ensemble de termes et $\mathcal{R}(\mathcal{T})$ l'ensemble qui constituerait *une seule étape* d'application de $\mathcal{R}$, soit $\mathcal{R}(\mathcal{T}) = \{t' \mid t \in \mathcal{T} \ et \ t \to_{\mathcal{R}} t'\}$. En général, la construction d'un automate reconnaissant $\mathcal{R}(\mathcal{L}(\mathcal{A}))$ pour automate $\mathcal{A}$ donné n'est pas simple. En particulier, l'algorithme de complétion (section 2.4) ne permet pas une telle chose, comme nous allons le voir dans l'exemple suivant.

EXEMPLE *3.10*
Soient $\mathcal{R} = \{f(x) \to g(x)\}$ un système de réécriture et $\mathcal{A} = \langle \mathcal{F}, \mathcal{Q}, \mathcal{Q}_f, \Delta \rangle$ un automate d'arbres tel que $\mathcal{Q} = \mathcal{Q}_f = \{q\}$ et $\Delta = \{f(q) \to q, a \to q\}$. Alors $\mathcal{L}(\mathcal{A}) = \{f^*(a)\}$ et $\mathcal{R}(\mathcal{L}(\mathcal{A})) = \{f^*(g(f^*(a)))\}$ (une seule étape : un seul f se réécrit en g). Cependant, si l'on applique l'algorithme de complétion pour $\mathcal{R}$ et $\mathcal{A}$, après la première étape on obtient l'automate $\mathcal{A}^1_{\mathcal{R}} = \mathcal{A} \cup \{g(q_0) \to q_0\}$. On a alors $\mathcal{L}(\mathcal{A}^1_{\mathcal{R}}) = \{(fg)^*(a)\} = \mathcal{R}^*(\mathcal{L}(\mathcal{A}))$, soit $\mathcal{L}(\mathcal{A}^1_{\mathcal{R}}) \supset \mathcal{R}(\mathcal{L}(\mathcal{A}))$. ◀

Cependant, des travaux modifiant l'algorithme de complétion permettent de vérifier des propriétés temporelles en utilisant la complétion d'automates d'arbres [Boyer et Genet, 2009]. L'idée est d'utiliser des ϵ-transitions spéciales, permettant dans un automate d'arbres de distinguer un terme de son successeur. Ainsi grâce à ses transitions, il est possible d'extraire un graphe orienté de l'historique du système, sur lequel il est possible de vérifier des propriétés temporelles de la logique LTL (*Linear Temporal Logic*) grâce à un algorithme de *Model-Checking* standard.

3.3 Fonctions d'abstractions du model-checking régulier : Abstract Regular (Tree) Model-Checking

3.3.1 Problématique et exemple fil rouge

Nous avons vu en section 2.4.2 que le calcul de l'ensemble des configurations accessibles était rarement convergent, notamment dans le cas de systèmes à états infinis. C'est pourquoi différentes méthodes d'abstractions ont été développées afin d'accélérer la convergence du calcul, et permettre ainsi de calculer une sur-approximation de l'ensemble des configurations accessibles en un temps fini. Soit $\mathcal{R}^*(\mathcal{L}(\mathcal{A}))$ l'ensemble des configurations accessibles d'un système, et $\mathcal{A}_{app}$ l'automate point-fixe calculé en utilisant une des méthodes d'abstraction que nous allons détailler par la suite, alors $\mathcal{L}(\mathcal{A}_{app}) \supseteq \mathcal{R}^*(\mathcal{L}(\mathcal{A}))$, *i.e.* le langage de l'automate point-fixe calculé en utilisant les méthodes d'abstractions est une sur-approximation de l'ensemble des configurations accessibles. Ainsi, pour la vérification d'une propriété de sûreté, soit Bad l'ensemble des configurations interdites, si $\mathcal{L}(\mathcal{A}_{app}) \cap Bad = \emptyset$, alors cela implique que $\mathcal{R}^*(\mathcal{L}(\mathcal{A})) \cap Bad = \emptyset$. Autrement dit, si une propriété de sûreté est vérifiée pour un sur-ensemble des comportements du programme, alors elle est également vraie pour le programme.

Dans le cadre de l'Abstract Regular (Tree) Model-Checking [Bouajjani et al., 2004, Bouajjani et al., 2006a], ou dans la complétion d'automates d'arbres (voir section 2.4.2), les méthodes d'abstractions cherchent à limiter voir réduire le nombre d'états de l'automate courant, à chaque étape de calcul du successeur (*i.e.* à chaque étape de complétion ou de calcul de $\tau(\mathcal{A}_i)$). Ainsi, si le nombre d'état devient borné, cela force la convergence du calcul des accessibles.

La plupart de ces techniques se basent donc sur la fusion de plusieurs états en un seul à chaque étape de calcul de l'automate successeur. Les états sont fusionnés selon certains critères d'équivalence que nous allons voir dans les sous-sections suivantes. C'est le cas également de l'abstraction équationnelle que nous avons vue en section 2.4.2, qui permet de fusionner des états équivalents selon un ensemble d'équations déterminées par l'utilisateur.

EXEMPLE *3.11*
Soient $\mathcal{R} = \{f(x) \to f(s(s(x)))\}$, $Bad = \{f(s(zero)), f(s(s(s(zero))))\}$, et $\mathcal{A}_0$ l'automate d'arbres possédant l'ensemble de transitions $\Delta_0 = \{zero \to q_0, f(q_0) \to q_f\}$, avec q_f l'état final.

Alors au bout d'une étape de complétion on obtient l'automate $\mathcal{A}_1$ possédant l'ensemble de transitions Δ_1 suivant :

$$\begin{aligned}\Delta_1 = \{ \quad & zero \to q_0, f(q_0) \to q_f \\ & s(q_0) \to q_1, s(q_1) \to q_2 \\ & f(q_2) \to q_f \ \}\end{aligned}$$

Puis au bout de 2 étapes de complétion on obtient l'automate $\mathcal{A}_2$ possédant l'ensemble de transitions Δ_2 suivant :

$$\begin{aligned}\Delta_2 = \{ \quad & zero \to q_0, f(q_0) \to q_f \\ & s(q_0) \to q_1, s(q_1) \to q_2 \\ & f(q_2) \to q_f \\ & s(q_2) \to q_3, s(q_3) \to q_4 \\ & f(q_4) \to q_f \ \}\end{aligned}$$

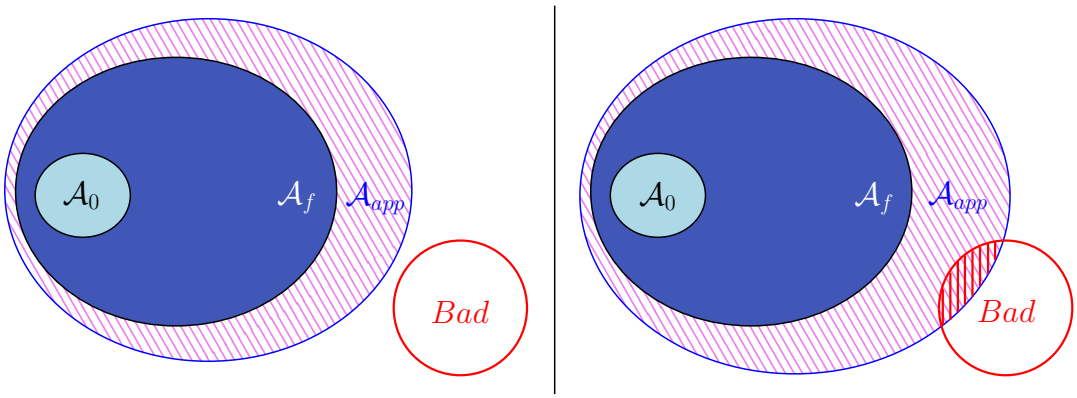


FIGURE 3.6 — Cas 1 : l'intersection entre la sur-approximation et l'ensemble des termes interdits est vide, la propriété est donc vérifiée. Cas 2 : la sur-approximation est trop grossière, on a un *faux contre-exemple*.

Soit $E = \{s(x) = s(s(s(x)))\}$ un ensemble d'équations. Alors les états q_1 et q_3 sont fusionnés, ainsi que les états q_2 et q_4. Soit $\mathcal{A}_2 \rightsquigarrow_E \mathcal{A}'_2$, alors $\mathcal{A}'_2$ possède l'ensemble de transitions Δ'_2 suivant :

$$\begin{aligned}\Delta'_2 = \{ \quad & zero \to q_0, f(q_0) \to q_f \\ & s(q_0) \to q_1, s(q_1) \to q_2 \\ & f(q_2) \to q_f \\ & s(q_2) \to q_1 \ \}\end{aligned}$$

Pour tout $q \in \mathcal{A}'_2$, on a $f(q) \to q' \in \Delta'_2 \implies f(s(s(q))) \to^*_{\mathcal{A}'_2} q'$. L'automate $\mathcal{A}'_2$ est donc $\mathcal{R}$-clos et point-fixe. Par ailleurs, on a également $\mathcal{L}(\mathcal{A}'_2) \cap Bad = \emptyset$ (rappelons que q_f est le seul état final). ◀

On remarque dans cet exemple que le choix de l'équation est essentiel dans l'approximation. En effet, si l'équation choisie avait été $f(s(x)) = f(s(s(x))$, alors les états q_1 et q_2 auraient été fusionnés, ainsi que les états q_3 et q_4. L'ensemble de transitions Δ'_2 aurait donc été le suivant :

$$\begin{aligned}\Delta'_2 = \{ \quad & zero \to q_0, f(q_0) \to q_f \\ & s(q_0) \to q_1, s(q_1) \to q_1 \\ & f(q_1) \to q_f \\ & s(q_1) \to q_3, s(q_3) \to q_3 \\ & f(q_3) \to q_f \ \}\end{aligned}$$

Cet automate est $\mathcal{R}$-clos. Cependant, on a $f(s(zero)) \to_{\mathcal{A}'_2} f(s(q_0)) \to_{\mathcal{A}'_2} f(q_1) \to_{\mathcal{A}'_2} q_f$, *i.e.* $f(s(zero)) \in \mathcal{L}(\mathcal{A}'_2)$, soit $\mathcal{L}(\mathcal{A}'_2) \cap Bad \neq \emptyset$. Or, comme nous l'avons vu dans l'exemple 3.11 avec la bonne équation d'approximation, le système vérifie bien la propriété, *i.e.* n'accepte aucun terme interdit. Ce contre-exemple n'a donc été induit que par la sur-approximation. Nous avons donc ici un **faux contre-exemple** dû à une fonction d'approximation mal choisie. Ce problème du faux contre-exemple est résumé dans la figure 3.6.

Pour palier à ce problème, afin que la sur-approximation calculée n'admette plus de faux contre-exemples, la fonction d'abstraction doit donc être modifiée : on appelle alors ceci le **raffinement** de l'abstraction. Dans le cas de l'abstraction équationnelle, il s'agit par exemple d'annuler des fusions d'états induites par les équations.

Nous allons donc comparer dans cette section la méthode d'abstraction par équation liée à la complétion [Genet et Rusu, 2010] et les méthodes d'abstractions de l'*Abstract Regular Tree Model-Checking* [Bouajjani et al., 2006a] (où la fonction de transition est donc un transducteur d'arbres). Il est à noter que ces fonctions d'abstractions sont identiques à celles de l'*Abstract Regular Model-Checking* (cas des automates de mots). Nous allons ensuite décrire les différentes méthodes d'abstractions utilisées par la complétion d'automates d'arbres, notamment dans l'outil Timbuk, en plus de l'abstraction équationnelle (sous-section 3.3.4).

Il existe également d'autres méthodes d'abstractions, que nous allons décrire en sous-section 3.3.5, qui permettent d'accélérer le calcul des accessibles sans utiliser la fusion d'états, mais en utilisant des techniques de *widening* sur la structure des automates d'arbres après chaque étape de complétion, ou cherchant à inférer automatiquement des équations d'après la structure du système de réécriture.

Nous verrons enfin des solutions proposées pour le raffinement de chacune de ces abstractions, ainsi que des méthodes de raffinement automatique dans le cas de l'*Abstract Regular (Tree) Model-Checking* et de la complétion d'automates d'arbres.

L'exemple 3.11 sera utilisé tout au long de cette section afin d'appliquer et comparer les différentes méthodes d'abstractions décrites.

3.3.2 Fusion d'états par équivalence de langage jusqu'à une hauteur n

Cette méthode d'abstraction, décrite dans [Bouajjani et al., 2006a], calcule le langage de chaque état de l'automate et les compare selon leur "hauteur". La hauteur d'un terme t correspond à la distance entre sa racine et sa feuille la plus profonde. Par exemple, $hauteur(f(g(a), g(g(b))) = 4$. Soit $\mathcal{A}$ un automate d'arbres et q un état de $\mathcal{A}$, alors $\mathcal{L}^{\leq n}(\mathcal{A}, q)$ contient les termes reconnus par l'état q dont la hauteur est inférieure ou égale à n, *i.e.* $\mathcal{L}^{\leq n}(\mathcal{A}, q) = \{t \in \mathcal{L}(\mathcal{A}, q) \mid hauteur(t) \leq n\}$.

Deux états q_1 et q_2 sont alors dits équivalents si leur langage jusqu'à une hauteur n est le même, *i.e.* si $\mathcal{L}^{\leq n}(\mathcal{A}, q_1) = \mathcal{L}^{\leq n}(\mathcal{A}, q_2)$. Dans ce cas, les états q_1 et q_2 sont alors fusionnés dans $\mathcal{A}$ (voir définition 2.4.8).

EXEMPLE *3.12*
Reprenons l'automate $\mathcal{A}_2$ de l'exemple 3.11. On a alors :
$\mathcal{L}(\mathcal{A}, q_1) = \{s(zero)\}$, $\mathcal{L}(\mathcal{A}, q_2) = \{s(s(zero))\}$,
$\mathcal{L}(\mathcal{A}, q_3) = \{s(s(s(zero)))\}$, $\mathcal{L}(\mathcal{A}, q_4) = \{s(s(s(s(zero))))\}$.

Considérons une hauteur $n = 3$. Alors dans ce cas, on a :
$\mathcal{L}^{\leq 3}(\mathcal{A}, q_1) = \{s(zero)\}$, $\mathcal{L}^{\leq 3}(\mathcal{A}, q_2) = \{s(s(zero))\}$,
$\mathcal{L}^{\leq 3}(\mathcal{A}, q_3) = \emptyset$, $\mathcal{L}^{\leq 3}(\mathcal{A}, q_4) = \emptyset$.

On a alors $\mathcal{L}^{\leq 3}(\mathcal{A}, q_3) = \mathcal{L}^{\leq 3}(\mathcal{A}, q_4)$, ce qui implique la fusion des états q_3 et q_4. Après fusion de ces deux états, on obtient l'automate $\mathcal{A}'_2$ possédant l'ensemble de transitions suivant :

$$\begin{aligned}\Delta'_2 = \{ \quad & zero \to q_0, f(q_0) \to q_f \\ & s(q_0) \to q_1, s(q_1) \to q_2 \\ & f(q_2) \to q_f \\ & s(q_2) \to q_3, s(q_3) \to q_3 \\ & f(q_3) \to q_f \ \}\end{aligned}$$

Cet automate reconnait le langage suivant :

$$\mathcal{L}(\mathcal{A}'_2) = \{f(s(s(zero))), f(s(s(s(zero)))), f(s(s(s(s^*(zero)))))\}.$$

$\mathcal{A}'_2$ est donc $\mathcal{R}$-clos. Cependant $\mathcal{L}(\mathcal{A}'_2) \cap Bad \neq \emptyset$, on est donc en présence d'un faux contre-exemple, cette approximation n'est pas assez fine. ◀

Dans le cas d'une approximation trop grossière, le raffinement proposé pour cette technique d'abstraction est d'augmenter la hauteur pour l'équivalence de langage. Dans l'exemple 3.12 ci-dessus, on peut par exemple prendre $n = 4$. Seulement, à cette étape du calcul, aucun état ne sera fusionné et $\mathcal{A}'_2$ ne sera pas $\mathcal{R}$-clos.

EXEMPLE *3.13*
Reprenons l'automate $\mathcal{A}_3$ après une 3e étape de complétion. On a alors l'ensemble de transitions suivant :

$$\begin{aligned}\Delta_3 = \{ \quad & zero \to q_0, f(q_0) \to q_f \\ & s(q_0) \to q_1, s(q_1) \to q_2 \\ & f(q_2) \to q_f \\ & s(q_2) \to q_3, s(q_3) \to q_4 \\ & f(q_4) \to q_f, \\ & s(q_4) \to q_5, s(q_5) \to q_6 \\ & f(q_6) \to q_f \ \}\end{aligned}$$

Pour une hauteur $n = 4$, on a alors $\mathcal{L}^{\leq 4}(\mathcal{A}, q_4) = \emptyset = \mathcal{L}^{\leq 4}(\mathcal{A}, q_5) = \mathcal{L}^{\leq 4}(\mathcal{A}, q_6)$. Les états q_4, q_5 et q_6 sont alors fusionnés et on obtient alors l'ensemble de transitions suivant :

$$\begin{aligned}\Delta'_3 = \{ \quad & zero \to q_0, f(q_0) \to q_f \\ & s(q_0) \to q_1, s(q_1) \to q_2 \\ & f(q_2) \to q_f \\ & s(q_2) \to q_3, s(q_3) \to q_4 \\ & f(q_4) \to q_f, \\ & s(q_4) \to q_4 \ \}\end{aligned}$$

On a alors $\mathcal{L}(\mathcal{A}'_3) = \{f(s(s(zero))), f(s(s(s(s(zero))))), f(s(s(s(s(s(s^*(zero)))))))\}$. On a donc bien $\mathcal{L}(\mathcal{A}'_3) \cap Bad = \emptyset$. ◀

Cependant, si l'ensemble Bad contenait tous les nombres impairs, (autrement dit $Bad = \{f([s(s(]^*s(zero))^*\})$, alors l'abstraction par équivalence de langage jusqu'à une hauteur n ne serait pas pertinente sur cet exemple, puisqu'elle ne fusionnerait que deux états consécutifs et ne pourrait donc pas séparer les nombres pairs des nombres impairs.

3.3.3 Fusion d'états par équivalence de prédicats

Nous allons désormais décrire l'autre méthode d'abstraction énoncée dans [Bouajjani et al., 2006a], basée sur un calcul de prédicats d'*arbres*. Soit $\mathcal{P} = \{P_1, \ldots, P_n\}$ un ensemble de *prédicats* tel que chaque prédicat $P \in \mathcal{P}$ est un langage d'arbres représenté par un automate d'arbres. Soient $\mathcal{A}$ un automate d'arbres et q_1, q_2 deux états de $\mathcal{A}$. Alors

q_1 et q_2 sont dits équivalents si l'intersection entre leurs langages et l'ensemble $\mathcal{P}$ est non vide et contient le même sous-ensemble de prédicats. Formellement, q_1 et q_2 sont équivalents s'ils respectent la condition suivante :

$$\forall P \in \mathcal{P}, \mathcal{L}(P) \cap \mathcal{L}(\mathcal{A}, q_1) = \emptyset \Leftrightarrow \mathcal{L}(P) \cap \mathcal{L}(\mathcal{A}, q_2) = \emptyset.$$

Notons que l'ensemble $\mathcal{P}$ doit toujours posséder un nombre fini de prédicats. Cette fonction d'abstraction est raffinée en ajoutant des prédicats à l'ensemble $\mathcal{P}$. Le principal intérêt de cette méthode est que si l'ensemble $\mathcal{P}$ de prédicats est construit à partir de l'ensemble Bad, alors cela peut éliminer les éventuels faux contre-exemples.

THÉORÈME 3.3.1
Soient $\mathcal{A} = \langle \mathcal{F}, \mathcal{Q}, \mathcal{Q}_f, \Delta \rangle$ et $\mathcal{A}_{Bad} = \langle \mathcal{F}, \mathcal{Q}_{Bad}, \mathcal{Q}_{f_{Bad}}, \Delta_{Bad} \rangle$ deux automates d'arbres, et $\mathcal{P}$ un ensemble de prédicats tel que $\forall q_{Bad} \in \mathcal{Q}_{Bad}, \exists P \in \mathcal{P}$ t.q. $\mathcal{L}(\mathcal{A}_{Bad}, q_{Bad}) = \mathcal{L}(P)$. Soit $\mathcal{A}'$ l'automate d'arbres après abstraction par l'ensemble de prédicats . Alors, si $\mathcal{L}(\mathcal{A}) \cap \mathcal{L}(\mathcal{A}_{Bad}) = \emptyset$, $\mathcal{L}(\mathcal{A}') \cap \mathcal{L}(\mathcal{A}_{Bad}) = \emptyset$ également.

EXEMPLE *3.14*
Rappelons que $Bad = \{f(s(zero)), f(s(s(s(zero))))\}$. Si l'on choisit d'étendre cet ensemble à celui des nombres impairs, *i.e.* $f([s(s(]^* s(zero))$, on obtient alors l'automate d'arbres $\mathcal{A}_{Bad}$ possédant l'ensemble de transitions suivant, où q'_f est le seul état final :

$$\begin{aligned}\mathcal{A}_{Bad} = \{ \quad & zero \to q'_0, s(q'_0) \to q'_1, \\ & s(q'_1) \to q'_2, s(q'_2) \to q'_1, f(q'_1) \to q'_f \ \}\end{aligned}$$

Si l'on construit l'ensemble de prédicats comme indiqué dans le théorème 3.3.1, on obtient alors l'ensemble $\mathcal{P} = \{P_0, P_1, P_2, P_3\}$, où $P_0, \ldots, P_3$ sont les automates d'arbres suivants.

Prédicat	Ensemble de transitions	États finaux	Langage du prédicat
P_0	$\Delta_{P_0} = \{zero \to q_{P_0}\}$	q_{P_0}	$\mathcal{L}(P_0) = \{zero\}$
P_1	$\Delta_{P_1} = \Delta_{P_0} \cup \{s(q_{P_0}) \to q_{P_1}, s(q_{P_1}) \to q_{P_2}, s(q_{P_2}) \to q_{P_1}\}$	q_{P_1}	$\mathcal{L}(P_1) = \{[s(s(]^* s(zero))\}$
P_2	$\Delta_{P_2} = \Delta_{P_1}$	q_{P_2}	$\mathcal{L}(P_2) = \{[s(s(]^* s(s(zero))))\}$
P_3	$\Delta_{P_3} = \Delta_{P_1} \cup \{f(q_{P_1}) \to q_{P_3}\}$	q_{P_3}	$\mathcal{L}(P_3) = \{f([s(s(]^* s(zero))\}$

Reprenons l'automate $\mathcal{A}_2$ de l'exemple fil rouge 3.11. On a alors :
$\mathcal{L}(\mathcal{A}, q_0) = \{zero\}$, $\mathcal{L}(\mathcal{A}, q_f) = \{f(s(s(zero))), f(s(s(s(s(zero)))))\}$,
$\mathcal{L}(\mathcal{A}, q_1) = \{s(zero)\}$, $\mathcal{L}(\mathcal{A}, q_2) = \{s(s(zero))\}$,
$\mathcal{L}(\mathcal{A}, q_3) = \{s(s(s(zero)))\}$, $\mathcal{L}(\mathcal{A}, q_4) = \{s(s(s(s(zero))))\}$.

Calculons maintenant l'intersection entre ces langages et ceux des prédicats de l'ensemble $\mathcal{P}$:

$\cap$	P_0	P_1	P_2	P_3
$\mathcal{L}(\mathcal{A}, q_0)$	$\neq \emptyset$	$= \emptyset$	$= \emptyset$	$= \emptyset$
$\mathcal{L}(\mathcal{A}, q_1)$	$= \emptyset$	$\neq \emptyset$	$= \emptyset$	$= \emptyset$
$\mathcal{L}(\mathcal{A}, q_2)$	$= \emptyset$	$= \emptyset$	$\neq \emptyset$	$= \emptyset$
$\mathcal{L}(\mathcal{A}, q_3)$	$= \emptyset$	$\neq \emptyset$	$= \emptyset$	$= \emptyset$
$\mathcal{L}(\mathcal{A}, q_4)$	$= \emptyset$	$= \emptyset$	$\neq \emptyset$	$= \emptyset$
$\mathcal{L}(\mathcal{A}, q_f)$	$= \emptyset$	$= \emptyset$	$= \emptyset$	$= \emptyset$

Constatons tout d'abord que les langages des états q_1 et q_3 ont une intersection non vide avec le même sous-ensemble de $\mathcal{P}$ (ici P_1). Il en est de même pour q_2 et q_4 (prédicat P_2). Les états q_1 et q_3 sont donc fusionnés, ainsi que les états q_2 et q_4, et on obtient l'automate $\mathcal{A}'_2$ après fusion, possédant l'ensemble de transitions suivant.

$$\begin{aligned}\Delta'_2 = \{ \quad & zero \to q_0, f(q_0) \to q_f \\ & s(q_0) \to q_1, s(q_1) \to q_2 \\ & f(q_2) \to q_f \\ & s(q_2) \to q_1 \ \}\end{aligned}$$

Cet automate est $\mathcal{R}$-clos, et on a $\mathcal{L}(\mathcal{A}'_2) = \{f([s(s(]^* zero)\}$. Cet automate vérifie donc $\mathcal{L}(\mathcal{A}'_2) \cap Bad = \emptyset$. ◀

3.3.4 Autres fonctions d'abstractions de l'outil Timbuk

Normalisation par des fonctions d'abstractions

Cette méthode d'abstraction, décrite dans [Genet et Viet Triem Tong, 2001a], a été implémentée dans Timbuk avant l'abstraction équationnelle. Elle consiste, au moment de la normalisation des transitions à ajouter dans l'automate successeur, à utiliser des règles de normalisation définissant une abstraction (*i.e.* utilisant des états existants plutôt que de nouveaux états).

La forme générale de ces règles de normalisation est la suivante : $[s \to x] \to [l_1 \to x_1, \ldots, l_n \to x_n]$, avec $s \in \mathcal{T}(\mathcal{F} \cup \mathcal{Q}, \mathcal{X})$ et $x \in \mathcal{X} \cup \mathcal{Q}$, où $[s \to x]$ est un *pattern* qui doit être filtré pour correspondre aux transitions à normaliser, et où $[l_1 \to x_1, \ldots, l_n \to x_n]$ sont les règles utilisées pour normaliser la partie gauche de la nouvelle transition. Pour normaliser une transition de la forme $t \to q'$, on filtre s sur t, et x sur q'. On obtient alors une substitution σ qui permet de normaliser $t \to q'$ avec les règles de réécriture $l_1\sigma \to \sigma(x_1), \ldots, l_n\sigma \to \sigma(x_n)$, où $\sigma(x_1), \ldots, \sigma(x_n)$ doivent correspondre à des états.

EXEMPLE *3.15*
Reprenons l'automate $\mathcal{A}_1$ de l'exemple 3.11. L'étape de complétion rajoute alors la transition $f(s(s(q_2))) \to q_f$ qu'il faut normaliser. Si l'on a la règle de normalisation suivante : $[f(s(s(x))) \to y] \to [s(s(x)) \to x, f(x) \to y]$, on obtient alors la substitution $\sigma = \{x \mapsto q_2, y \mapsto q_f\}$.

Soit q_3 un nouvel état, la transition $f(s(s(q_2))) \to q_f$ est alors normalisée par les transitions $s(q_2) \to q_3, s(q_3) \to q_2$ et $f(q_2) \to q_f$. On obtient alors l'automate $\mathcal{A}'_2$ possédant l'ensemble de transitions suivant :

$$\begin{aligned}\Delta'_3 = \{ \quad & zero \to q_0, f(q_0) \to q_f \\ & s(q_0) \to q_1, s(q_1) \to q_2 \\ & f(q_2) \to q_f \\ & s(q_2) \to q_3, s(q_3) \to q_2 \ \}\end{aligned}$$

Le langage reconnu par cet automate est $\mathcal{L}(\mathcal{A}'_2) = \{f([s(s(]^* zero)\}$, ainsi $\mathcal{A}'_2$ est $\mathcal{R}$-clos et on a bien $\mathcal{L}(\mathcal{A}'_2) \cap Bad = \emptyset$. ◀

Équations contextuelles

Cette méthode d'approximation, présente dans [Genet, 2008] et décrite dans [Genet, 2009], permet d'appliquer un ensemble d'équations d'abstractions, mais seulement dans un certain contexte. Imaginons par exemple l'équation $s(x) = s(y)$. On souhaite que

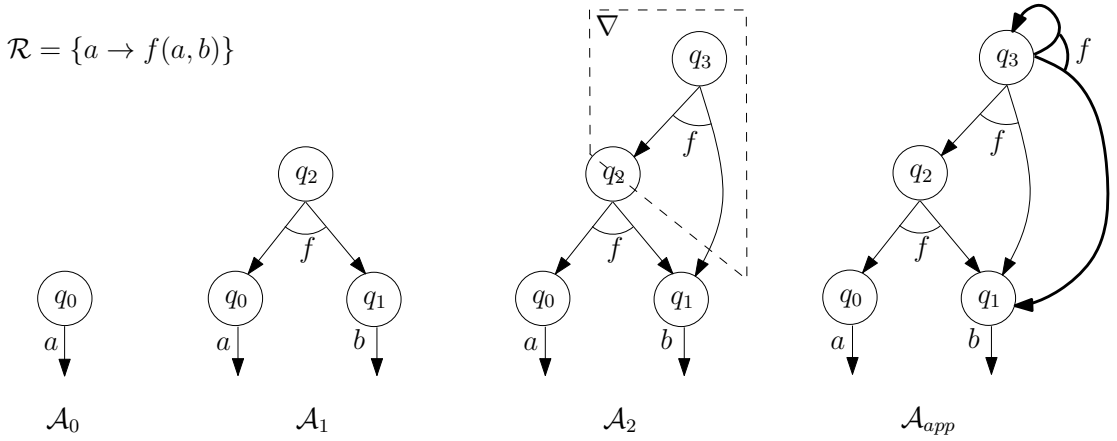


FIGURE 3.7 — Détection d'une situation de *widening* (∇) et calcul d'une sur-approximation $\mathcal{A}_{app}$ (figure tirée de [Bouajjani et Touili, 2002]).

cette équation ne s'applique que si les termes $s(x)$ et $s(y)$ se situent sous un f. Alors l'équation contextuelle adéquate sera de cette forme : $[f(s(x)), f(s(y))] \Rightarrow [s(x) = s(y)]$.

Les équations contextuelles peuvent être de trois formes différentes :

1. $[s] \Rightarrow [s_1 = t_1, \ldots, s_n = t_n]$,
2. $[s, t] \Rightarrow [s_1 = t_1, \ldots, s_n = t_n]$ et
3. $[s = t] \Rightarrow [s_1 = t_1, \ldots, s_n = t_n]$.

avec $s, t, s_1, \ldots, s_n, t_1, \ldots, t_n$ des termes de $\mathcal{T}(\mathcal{F}, \mathcal{X})$.

EXEMPLE *3.16*
Prenons notre automate $\mathcal{A}_3$ de l'exemple fil rouge et l'équation contextuelle $[f(s(x)) \Rightarrow [s(x) = s(s(s(x)))]$. Alors que l'équation $s(x) = s(s(s(x)))$ sans contexte permet de fusionner les états q_1 et q_3, ainsi que les états q_2 et q_4, cette équation contextuelle ne va fusionner que l'état q_2 avec l'état q_4, car elle ne concerne que le terme $s(x)$ situé directement sous un f, *i.e.* qui se reconnait en q_2.

3.3.5 Autres techniques d'abstractions basées sur le *widening*

Techniques de *widening* sur les automates d'arbre

Dans [Bouajjani et Touili, 2002], les auteurs décrivent une technique de *widening* analysant la structure de deux automates successeurs. Ainsi, les automates d'arbres sont représentés par des hypergraphes orientés (définition 3.2.4). Cette technique consiste, en comparant les hypergraphes de deux automates successeurs, à détecter des sous-hypergraphes qui vont se répéter indéfiniment. Ainsi, grâce à la répétition trouvée, l'opérateur de *widening* permet de créer un automate représentant une sur-approximation des configurations accessibles en ajoutant une boucle sur l'hypergraphe le représentant, comme nous le voyons en figure 3.7.

Trouver des répétitions entre deux hypergraphes, *i.e.* détecter des situations de *widening*, est un problème NP-complet. Certaines stratégies de détection existent, mais elles sont incomplètes.

Inférence automatique d'équations

Depuis un graphe particulier (appelé *Generalized sticking-out graph*) permettant de représenter un système de réécriture $\mathcal{R}$, la méthode décrite dans [Takai, 2004] infère automatiquement un ensemble d'équations d'abstraction. Ces équations sont appliquées à chaque étape d'un algorithme de complétion adapté à cette méthode. Une fois appliquées, ces équations forcent la création de boucles dans l'automate courant.

Le grand avantage de cette méthode est qu'elle est entièrement automatique. Cependant, il s'agit essentiellement d'une intuition et aucune preuve formelle de la correction de cet algorithme n'est donnée. De plus, la technique décrite dans [Takai, 2004] ne fonctionne pas pour tous les systèmes de réécriture, même linéaires, et ne permet d'inférer qu'un ensemble limité d'équations. Par exemple, une équation du type $f(g(x)) = g(f(x))$ ne pourra pas être inférée.

À l'inverse, l'algorithme de complétion décrit dans cette thèse a été formellement prouvé correct, manuellement (voir théorème 2.4.11 du chapitre 2 et preuve dans [Genet et Rusu, 2010, Genet, 2009]) et via l'outil de preuve `Coq` [Boyer et al., 2008], mais nécessite que les équations soient indiquées manuellement par l'utilisateur. Il existe cependant une exception concernant cet algorithme dans le cas des protocoles de sécurité, où des techniques permettant d'inférer automatiquement des règles de normalisations ont été développées dans [Boichut et al., 2008b].

3.3.6 Raffinement automatique depuis un faux contre-exemple

Les méthodes d'approximations comportant un paramétrage entré par l'utilisateur, tel qu'un ensemble d'équations [Genet et Rusu, 2010], une hauteur de langage, un ensemble de prédicats [Bouajjani et al., 2006a], ou encore des règles de normalisations [Genet et Viet Triem Tong, 2001a], posent la question suivante : comment déterminer le bon paramétrage ? En effet, nous avons vu en 3.3.1 et en figure 3.6 que les sur-approximations peuvent mener à de fausses conclusions sur le programme vérifié, et conclure à l'existence d'un contre-exemple qui en réalité n'a été introduit que par la sur-approximation. Il s'agit d'un *faux* contre exemple dû à une approximation trop grossière qu'il est alors nécessaire de raffiner. Mais après le calcul d'une sur-approximation, lorsqu'on est face à un contre-exemple, il n'est pas possible de décider dans l'immédiat s'il s'agit d'un un vrai ou d'un faux contre-exemple.

Soit $C \subseteq Bad$ l'ensemble des contre-exemples rencontrés, alors l'approche à la CEGAR (*Counter Example Guided Abstraction Refinement*) consiste à retirer successivement chaque contre-exemple $c \in C$ par raffinement de l'abstraction,. Chaque étape de raffinement conduit alors à la reprise du calcul des accessibles privés de c. Cependant, si l'approximation est trop fine, cela peut conduire à une divergence du calcul des accessibles.

Raffinement par contre-exemple dans le cadre de l'*Abstract Regular (Tree) Model-Checking*

Dans [Bouajjani et al., 2004] (pour les automates de mots) et dans [Bouajjani et al., 2006a] (pour les automates d'arbres), les auteurs ont introduit une technique inspirée de l'approche CEGAR afin de détecter les faux contre-exemples et raffiner l'approximation. Rappelons que dans le cas du calcul d'une sur-approximation, à l'aide d'une

fonction d'abstraction que l'on notera α, chaque étape de calcul du successeur (*i.e.* $\tau(\mathcal{A}_i)$) est suivie d'une étape d'abstraction. En partant de $\mathcal{A}_0$, on calcule d'abord l'abstraction de $\mathcal{A}_0$, que l'on va noter $\alpha(\mathcal{A}_0)$, puis à partir de cette abstraction, on calcule ensuite l'automate successeur $\tau(\alpha(\mathcal{A}_0))$. L'automate successeur après i étapes de transduction est alors noté $\tau_\alpha^i(\mathcal{A}_0)$ avec $\tau_\alpha^i(\mathcal{A}_0) = \tau(\alpha(\mathcal{A}_{i-1}))$, comme nous pouvons le voir en figure 3.8. Nous notons également $\tau_\alpha^*(\mathcal{L}(\mathcal{A}_0))$ la sur-approximation de l'ensemble des accessibles calculée grâce à l'abstraction α (*i.e.* $\tau_\alpha^*(\mathcal{L}(\mathcal{A}_0)) \supseteq \tau^*(\mathcal{L}(\mathcal{A}_0))$). Alors si $\tau_\alpha^*(\mathcal{L}(\mathcal{A}_0)) \cap Bad \neq \emptyset$, cela signifie qu'il existe une suite :

$$\mathcal{A}_0, \tau_\alpha(\mathcal{A}_0), \tau_\alpha^2(\mathcal{A}_0), \ldots, \tau_\alpha^n(\mathcal{A}_0)$$

telle que $\mathcal{L}(\tau_\alpha^n(\mathcal{A}_0)) \cap Bad \neq \emptyset$.

Soit $X_n = \mathcal{L}(\tau_\alpha^n(\mathcal{A}_0)) \cap Bad$ l'ensemble des contre-exemples reconnus dans $\tau_\alpha^n(\mathcal{A}_0)$. Alors on cherche dans tous les automates précédents quels termes ont permis de produire les termes de X_n par τ. Pour cela, on calcule un sur-ensemble des prédécesseurs possibles en appliquant la relation de transition inverse (τ^{-1}), et parmi ces prédécesseurs on recherche ceux qui étaient effectivement reconnus par l'automate $\tau_\alpha^{n-1}(\mathcal{A}_0)$. Ce calcul constitue alors l'ensemble de termes X_{n-1}, *i.e.* on a pour tout $k \geq 0$, $X_k = \mathcal{L}\tau_\alpha^k(\mathcal{A}_0) \cap \tau^{-1}(X_{k+1})$. Deux cas peuvent alors apparaître :

(a) On obtient $X_0 = \mathcal{L}(\mathcal{A}_0) \cap (\tau^{-1})^n(X_n) \neq \emptyset$, ce qui signifie qu'au moins un des termes de X_0, produisant un terme de X_n, est reconnu par l'automate initial. Nous sommes donc en présence d'un **vrai** contre-exemple et la propriété n'est donc pas vérifiée par le système.

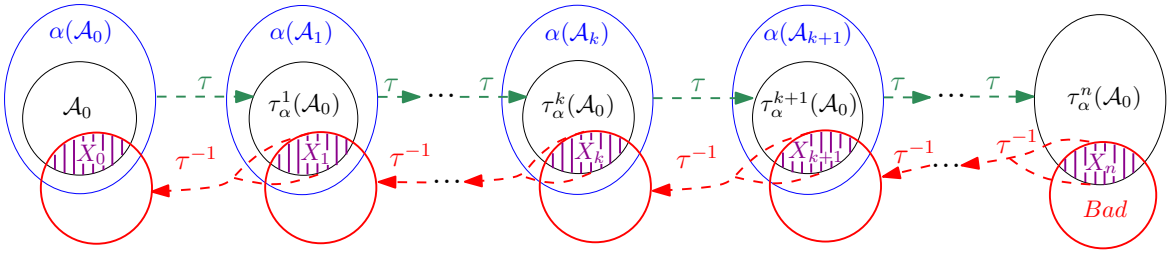
(b) À une étape k du processus ($0 \leq k \leq n$), on obtient $X_k = \emptyset$. On peut donc en déduire que les termes de X_{k+1} ont été produit par la sur-approximation.

Dans le cas 2, il faut alors raffiner l'abstraction afin que les termes de X^{k+1} ne soient plus acceptés par τ_α^{k+1}. Ce processus est résumé en figure 3.8 présentant les deux cas énumérés ci-dessus.

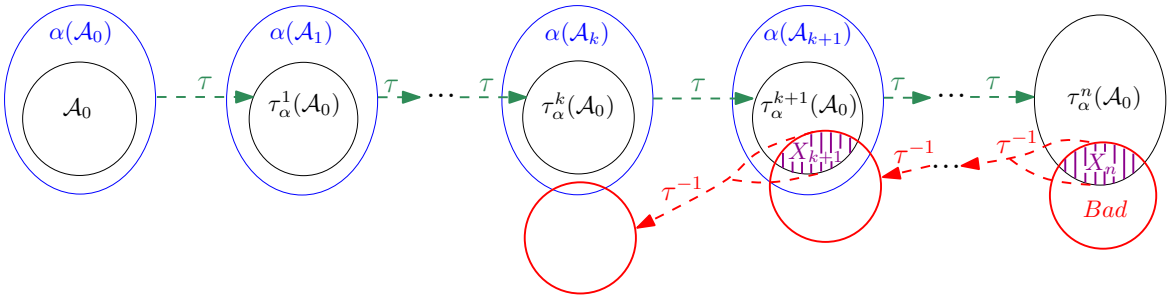
L'opération de détection du faux contre-exemple est assez coûteuse, et d'autant plus coûteuse que la propriété est violée tardivement. En effet, plus n est grand, plus la recherche en arrière (par application de τ^{-1}) est coûteuse, les opérations booléennes sur les automates d'arbres ayant un coût non négligeable.

Raffinement par contre-exemple pour la complétion d'automates d'arbres

Dans [Boichut et al., 2008a], les auteurs ont tenté une approche similaire à celle de [Bouajjani et al., 2006a] dans le cadre de la complétion d'automates d'arbres. Néanmoins, cette technique nécessite l'utilisation de règles de réécriture *linéaires* à droite et à gauche, pour pouvoir effectuer le calcul en arrière $\mathcal{R}^{-1}$ (pour un système de réécriture $\mathcal{R}$). En effet $\mathcal{R}^{-1}$ est défini en retournant chaque règle $l \to r$ de $\mathcal{R}$ en $r \to l$, imposant une linéarité des deux côtés pour que la complétion puissent être définie dans les deux sens (rappelons que l'algorithme de complétion n'est défini que pour des systèmes de réécriture linéaires *gauches*). Si la modélisation de systèmes par un ensemble de réécriture linéaires gauches est souvent immédiate, ce n'est pas toujours le cas lorsqu'on est limité à une linéarité des deux côtés.



(a) Réel contre-exemple.



(b) Faux contre-exemple : l'approximation doit être raffinée pour ne plus accepter aucun terme de X_{k+1}.

FIGURE 3.8 — Déterminer si le système admet un vrai ou un faux contre-exemple.

Une autre technique à la CEGAR a été développée et implémentée [Boichut et al., 2012], ne nécessitant cette fois pas de calcul en arrière, et par conséquent pas de linéarité à droite des règles de réécriture. Il s'agit d'un algorithme de complétion utilisant l'abstraction équationnelle, basé sur un nouveau type d'automates d'arbres appelés $\mathcal{R}_{/E}$-automates, gardant en mémoire une trace des équations utilisées durant le calcul de complétion. L'information nécessaire étant maintenue à jour dans le $\mathcal{R}_{/E}$-automate courant, il n'est plus nécessaire de revenir sur des calculs antérieurs pour connaitre la nature d'un terme t (*i.e.* vrai ou faux contre-exemple). La réduction du terme t dans l'automate permet d'obtenir des informations sur son atteignabilité, autrement dit permet de savoir si ce terme a été introduit ou non par des équations d'abstractions.

Les équations appliquées sont gardées en mémoire grâce à un nouveau type d'ϵ-transitions étiquetées par le symbole "=". Pour résumer, lorsque deux états q_1 et q_2 sont déclarés équivalents par équations, au lieu de les fusionner dans l'automate courant comme c'est le cas de la complétion classique, deux ϵ-transitions spéciales sont rajoutées à l'automate : $q_1 \overset{=}{\rightarrow} q_2$ et $q_2 \overset{=}{\rightarrow} q_1$. Cela équivaut à appliquer l'équation, mais tout en gardant une trace de cette dernière dans l'automate.

Ainsi, pour savoir si la réduction d'un terme est conditionnée ou non par des équations, il suffit de savoir si cette réduction utilise les ϵ-transitions étiquetées par le symbole "=".

EXEMPLE *3.17*
Reprenons l'automate $\mathcal{A}_1$ de l'exemple 3.11. Rappelons que son ensemble de transitions est le suivant :

$$\begin{aligned}\Delta_1 = \{ \quad & zero \rightarrow q_0, f(q_0) \rightarrow q_f \\ & s(q_0) \rightarrow q_1, s(q_1) \rightarrow q_2 \\ & f(q_2) \rightarrow q_f \ \}\end{aligned}$$

Soit $E = \{s(x) = s(s(x))\}$ un ensemble d'équations. Dans la complétion classique, on doit alors fusionner q_1 et q_2. Dans la $\mathcal{R}_{/E}$-complétion, on va rajouter les deux ϵ-transitions $q_1 \xrightarrow{=} q_2$ et $q_2 \xrightarrow{=} q_1$. Après abstraction, on a donc l'automate $\mathcal{A}'_1$ dont l'ensemble de transitions est $\Delta'_1 = \Delta_1 \cup \{q_1 \xrightarrow{=} q_2, q_2 \xrightarrow{=} q_1\}$.

Rappelons que $Bad = \{f(s(zero)), f(s(s(s(zero))))\}$. Or après abstraction, on a $f(s(zero)) \rightarrow_{\mathcal{A}'_1} f(s(q_0)) \rightarrow_{\mathcal{A}'_1} f(q_1) \xrightarrow{=}_{\mathcal{A}'_1} f(q_2) \rightarrow_{\mathcal{A}'_1} q_f$, donc $\mathcal{L}(\mathcal{A}'_1) \cap Bad \neq \emptyset$. Mais la réduction du terme interdit $f(s(zero))$ est due à l'abstraction, *i.e.* à la transition $q_1 \xrightarrow{=} q_2$ induite l'équation $q_1 = q_2$, ce qui est visible lors de la réduction du terme. La réduction d'un terme est donc conditionnée par l'existence des ϵ-transitions "=" induites par approximation, et ceci est noté $f(s(zero)) \xrightarrow{Eq(q_1,q_2)}{}^{*}_{\mathcal{A}'_1} q_f$. ◀

Ainsi, la réduction d'un terme est étiquetée par une condition qui permet immédiatement de connaître la nature du terme. Si, pour un terme t donné, il existe une réduction de ce terme dans l'automate qui ne soit pas conditionnée par une équation, alors c'est un vrai contre-exemple. Dans le cas contraire, on est en présence d'un faux contre-exemple. Dans ce cas, l'approximation doit être raffinée en enlevant une des équations conditionnant la réduction.

Cette technique à la $CEGAR$ sans calcul en arrière pour la complétion d'automates d'arbres a été implémentée dans un outil appelé TimbukCEGAR, une extention de Timbuk. Pour représenter les formules sur les équations (telles que par exemple $Eq(q_1, q_2) \wedge Eq(q_3, q_4)$), TimbukCEGAR exploite une représentation basée sur les BDD, utilisant la librairie *Buddy BDD* [Lind-Nielsen, 2002]. Les différentes expérimentations de l'outil TimbukCEGAR montrent que l'utilisation d'une librairie BDD augmente l'efficacité du calcul de $\mathcal{R}_{/E}$-complétion. Par conséquent, les performances de TimbukCEGAR sont les mêmes que Timbuk lorsqu'aucun raffinement n'est effectué, et les expérimentations menées donnent des résultats concluants. Dans les programmes testés, TimbukCEGAR surpasse généralement les autres implémentations CEGAR connues quand le raffinement est nécessaire.

3.4 Model-Checking avec Maude

Dans cette section nous allons décrire succinctement le fonctionnement de l'outil Maude, puis comment Maude permet de vérifier des systèmes à états infinis. Nous terminerons par une comparaison avec l'algorithme de complétion et Timbuk.

3.4.1 L'outil Maude

Maude [Clavel et al., 2009, Clavel et al., 2001] est un outil permettant de modéliser des systèmes par des règles de réécriture et/ou des équations. Il permet aussi de faire de la réécriture de terme, de l'analyse d'accessibilité d'un terme, ainsi que du Model-Checking. Le système Maude est une implémentation de la logique de réécriture et de la logique équationnelle (*Rewriting Logic* [Bruni et Meseguer, 2003], *Equational Logic* [Bouhoula et al., 2000]). Les modules Maude sont des programmes fonctionnels définissant une *théorie de réécriture*, qui permet de définir un langage de termes *typés*, ainsi qu'un ensemble d'équations et un ensemble de règles de réécriture. Dans une théorie de

réécriture, les termes sont construits en utilisant des opérateurs, qui sont des fonctions prenant zéro ou plusieurs arguments d'un certain type et retournant un terme typé.

EXEMPLE *3.18*
Soit `List` le type représentant une liste, et `Qid` le type représentant un élément d'une liste. Alors ces types sont déclarés de cette manière :

```
sorts List Qid .
```

Les opérateurs permettant de définir une liste sont déclarés de la manière suivante :

```
op nil : -> List .
(***Constante permettant de définir la liste vide.***)
op _._ : Qid List -> List .
(***Constructeur de liste : un élément en tête, puis le reste de la liste.
    Exemple : A . B . C . nil est une liste (A,B,C)***)
```

◀

Les opérateurs sont ensuite définis par :
- des équations, utilisées comme des règles de simplifications conduisant toujours au même résultat : il s'agit des calculs propres à un état du système, ou
- des règles de réécriture, décrivant les transitions entre les états du programme.

EXEMPLE *3.19*
Depuis l'exemple 3.18 précédent, on cherche à modéliser la fonction permettant de calculer la longueur d'une liste. Pour cela, déclarons tout d'abord un autre opérateur :

```
op long : List -> Nat .
(***Nat est un type existant représentant les entiers naturels.***)
```

Puis, avant de définir les équations `eq` permettant de calculer la longueur d'une liste, il faut au préalable déclarer les variables `var` qui seront utilisées dans les équations ou règles de réécriture :

```
var I : Qid .
var L : List .

eq long(nil) = 0
eq long(I . L) = s long(L) .
```

où `s_` est la fonction successeur prédéfinie dans le module des entiers (comprenant également le type `Nat`). ◀

Un module `Maude` est composé de déclarations de syntaxe, fournissant le langage approprié (et typé) pour décrire le système, et d'équations ou de règles de réécriture, décrivant le fonctionnement du système. La partie de déclaration de la syntaxe est appelée la *signature* et permet de déclarer :
- les types, donnant des noms aux types des données,
- les sous-types, permettant d'organiser les types sous forme de hiérarchie,
- les opérateurs, fournissant des noms aux opérations utilisées dans les équations et/ou les règles de réécriture,
- les variables, utilisées dans les équations et/ou les règles de réécriture.

Il existe deux types de modules : les modules fonctionnels et les modules systèmes. Les modules fonctionnels sont composés d'équations, et également de *memberships*, mais nous éluderons ces derniers dans cet état de l'art pour plus de simplicité (voir [Clavel et al., 2007]). Les modules systèmes, quant à eux, comportent des règles de réécriture et des équations.

3.4.2 Équations - Modules fonctionnels

Un module fonctionnel est encapsulé par les deux mots clés `fmod` et `endfm`. Les équations sont déclarées par le mot clé `eq` comme nous l'avons vu dans l'exemple 3.18. Il est également possible d'utiliser des équations *conditionnelles* via le mot clé `ceq` suivit d'une condition `if`, comme nous le voyons dans l'exemple ci-dessous.

EXEMPLE *3.20*
Le module suivant permet de définir le modulo entre deux entiers positifs.

```
fmod MODULO is
  protecting NAT .
  op _ modu _ : Nat Nat -> Nat .

  vars A B : Nat .

  ceq A modu B = sd(A,B) modu B if A >= B .
  ceq A modu B = A if A < B .

endfm
```

L'opérateur `sd(A,B)` effectue la différence symétrique (*i.e.* la valeur absolue de `A-B`). ◀

Certains attributs, tels que l'associativité, la commutativité ou l'idempotence peuvent être associés à chaque équation afin d'optimiser l'efficacité de leur exécution, que nous ne détaillerons et n'utiliserons pas dans cette section par soucis de simplification.

L'outil `Maude` effectue uniquement de la réécriture. Les équations sont en fait des règles de réécriture orientées de gauche à droite utilisées comme des règles de simplification. Le but des équations de `Maude`, contrairement à ses règles de réécriture, est d'atteindre un unique résultat final ou une forme canonique. Elles constituent en effet la partie de la théorie qui est supposée calculer toujours le même résultat, et en un temps fini. En effet, dans le cadre d'un système état-transition, elles concernent les calculs propres à un état, et pour appliquer la transition (qui sera représentée par un ensemble de règles de réécritures), il faut donc que les calculs à l'intérieur même d'un état soient terminant. Dans ce but, les équations doivent avoir certaines propriétés : être *confluentes*, *terminantes* et *sort-decreasing* (faire décroitre le type). Soit E l'ensemble d'équations d'un programme `Maude`, alors toute équation $t = t'$ de E est une règle de réécriture $t \to t'$. E étant un ensemble de règles de réécriture, les propriétés de confluence et de terminaison sont définies dans le chapitre 2 (définitions 2.1.21 et 2.1.19). Intuitivement, la décroissance de type (*sort-decreasiness*) signifie, pour un ensemble E d'équations supposé confluent et terminant, que la forme canonique d'un terme t par E doit avoir le plus petit type possible parmi les types de tous les termes lui étant équivalent par E. Rappelons que `Maude` permet le sous-typage. Il doit également être possible de

calculer ce plus petit type depuis la forme canonique de t, en utilisant uniquement les déclarations des opérateurs. Une définition plus formelle de la décroissance de type peut être trouvée dans [Bouhoula et al., 2000]. La propriété de terminaison peut être vérifiée par un outil de `Maude` appelé le *`Maude` terminaison tool* [Durán et al., 2008]. Les propriétés de confluence et de décroissance de type sont réunies dans une propriété appelée *Church-Rosser*, qui peut être vérifiée par un outil de `Maude` : le *Church-Rosser checker* [Durán et Meseguer, 2010a]. Il est possible de ne vérifier ces propriétés que pour les termes clos (***ground*** *Church-Rosser terminating*).

Une fois un module fonctionnel défini, on l'exécute en lui demandant de réduire un terme passé en entrée. Pour cela on utilise la commande :

```
reduce [option : nombre d'étapes de réduction] in <nom du module> : <terme>.
```

Si le module en question vient d'être chargé, cette commande peut être abrégée en `red <terme>`. Si aucune option sur le nombre d'étapes de réduction n'est mise, alors le terme est réduit jusqu'à sa forme canonique.

EXEMPLE *3.21*
Reprenons le module de l'exemple 3.20 calculant le modulo d'un nombre. Imaginons que l'on souhaite calculer le modulo de 8 par 3. Il suffit alors de réduire le terme `8 modu 3` après avoir chargé le module.

```
Maude> load modulo.maude
Maude> red 8 modu 3 .
reduce in MODULO : 8 modu 3 .
rewrites: 9 in 0ms cpu (0ms real) (160714 rewrites/second)
result NzNat: 2
```

◀

3.4.3 Règles de réécritures - Modules système

Introduisons maintenant les règles de réécriture de `Maude`, permettant de représenter le comportement d'un système, les équations représentant alors les axiomes du modèles (ou des règles de simplifications), ou encore des équations d'abstractions comme nous allons le voir en sous-section suivante. Les modules systèmes sont encapsulés entre les deux mots clés `mod` et `endm`, et une règle de réécriture s'introduit grâce au mot clé `rl` (ou `crl` dans le cas d'une règle de réécriture conditionnelle).

EXEMPLE *3.22*
Le protocole du boulanger, introduit en section 2.5, se représente d'une manière similaire sous `Maude` par le module suivant.

```
mod BAKERY is
 protecting NAT .
 sorts Mode BState .

 ops sleep wait crit : -> Mode .
 op state : Mode Nat Mode Nat -> BState .
 op initial : -> BState .
```

```
 vars P Q : Mode .
 vars X Y : Nat .

 eq initial = state(sleep, 0, sleep, 0) .

 rl  state(sleep, X, Q, Y) => state( wait, s Y, Q, Y) .
 rl  state(wait, X, Q, 0)  => state(crit, X, Q, 0) .
 crl state(wait, X, Q, Y)  => state(crit, X, Q, Y) if not (Y < X) .
 rl  state(crit, X, Q, Y)  => state(sleep, 0, Q, Y) .
 rl  state(P, X, sleep, Y) => state(P, X, wait, s X) .
 rl  state(P, 0, wait, Y)  => state(P, 0, crit, Y) .
 crl state(P, X, wait, Y)  => state(P, X, crit, Y) if Y < X .
 rl  state(P, X, crit, Y)  => state(P, X, sleep, 0) .

endm
```

◀

Pour les modules système, si des équations sont présentes en plus des règles de réécriture, la partie équationnelle doit être *Church-Rosser* et terminante comme nous l'avons vu dans la sous-section précédente, mais une autre condition doit être également réalisée. La stratégie adoptée par `Maude` dans le cas d'un ensemble d'équations E et d'un ensemble de règles de réécriture $\mathcal{R}$, est de tout d'abord appliquer E afin d'atteindre une forme canonique, puis ensuite effectuer une étape de réécriture avec une des règles de réécriture de $\mathcal{R}$, et ainsi de suite. Pour être certain que cette stratégie est complète, *i.e.* qu'elle n'entraine pas, à cause de l'étape de calcul de forme canonique par E, la non-application de certaines règles de réécriture de $\mathcal{R}$ qui l'auraient été sans application de E, il faut que le module respecte la propriété suivante : que l'ensemble E d'équations soit *cohérent* avec $\mathcal{R}$.

DÉFINITION 3.4.1 (Cohérence entre $\mathcal{R}$ et E)
Soient $\mathcal{R}$ l'ensemble de règles de réécriture d'un module M, E l'ensemble des équations de M et t un terme. On note $t \to^1_{\mathcal{R}} t'$ la réécriture de t en t' en une seule étape de réécriture (i.e. $t \to_{\mathcal{R}} t'$). On note $t \to^!_E u$ lorsque u est le terme canonique obtenu après réécriture de t via les équations.

Alors pour tout terme t tel que $t \to^1_{\mathcal{R}} t'$ et $t \to^!_E u$, il existe un u' tel que $u \to^1_{\mathcal{R}} u'$ et $t' =_E u'$, ce qui signifie, d'après la propriété Church-Rosser, que t' et u' possèdent la même forme canonique via E, i.e. qu'il existe un terme w tel que $t' \to^!_E w$ et $u' \to^!_E w$

Cette propriété est résumée en figure 3.9, et peut être démontrée pour un module donné grâce à l'outil de `Maude` appelé le *`Maude` Coherence Checker* [Durán et Meseguer, 2010b]. La vérification de cette propriété peut également se limiter au termes clos (***ground*** *coherence*).

Une fois un module système défini, on l'exécute en lui demandant de *réécrire* un terme passé en entrée. Pour cela on utilise la commande :

```
rewrite [option : nombre d'étapes de réduction] in <nom du module> : <terme>.
```

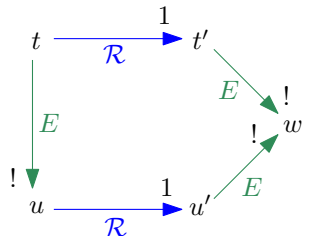


FIGURE 3.9 — Cohérence entre $\mathcal{R}$ et E.

Si le module en question vient d'être chargé, cette commande peut être abrégée en `rew` `<terme>`. Si aucune option sur le nombre d'étapes de réduction n'est mise, alors le terme est réduit jusqu'à sa forme canonique.

EXEMPLE *3.23*
Reprenons le module de l'algorithme du boulanger (exemple 3.22). Comme l'algorithme du boulanger peut se répéter à l'infini, nous allons appeler la commande `rewrite` avec un nombre limité d'étapes de réécriture (nous verrons dans la section suivante comment résoudre ce problème par une abstraction du calcul).

```
Maude> load Bakery/bakery.maude
Maude> rew [100] initial .
rewrite [100] in BAKERY : initial .
rewrites: 101 in 0ms cpu (0ms real) (101000000 rewrites/second)
result BState: state(sleep, 0, wait, 1)
```

◀

3.4.4 Model-Checking

Nous avons vu qu'il est possible d'exécuter un module à partir d'un terme (commande `reduce` ou `rewrite`). Mais pour vérifier des propriétés d'atteignabilité (ce terme est-il accessible ?) ou de sûreté (ce terme interdit ne peut pas être atteint), il faut utiliser la commande `search`, qui va permettre d'explorer l'espace des termes accessibles de différentes manières. Sa syntaxe est la suivante :

```
search [n,m] in <module> : <terme 1> <flèche de recherche> <terme 2> such that
                                <condition> .
```

où :
- `n` est un argument optionnel fournissant une borne sur le nombre de solutions désirées,
- `m` est un autre argument optionnel permettant de donner la profondeur maximum de la recherche,
- `<module>` est le module où la recherche est effectuée (il peut être omis s'il n'y a pas d'ambiguïté sur le module à utiliser),
- `<terme 1>` est le terme de départ,
- `<terme 2>` le *pattern* que l'on veut trouver, et
- `<flèche de recherche>` une flèche indiquant la forme de la réécriture que l'on veut utiliser. En voici deux exemples :

  - `=>1` effectue une seule étape de réécriture (*i.e.* permet de répondre à la question : ce terme est-il atteignable en une seule étape de réécriture ?)
  - `=>*` effectue toutes les étapes de réécriture possibles (*i.e.* permet de répondre à la question : ce terme fait-il parti de l'ensemble des termes atteignables ?)
- `<condition>` une condition booléenne à respecter pour le pattern `<terme 2>` à atteindre.

EXEMPLE *3.24*
La modélisation d'une horloge simple marquant les heures de la journée peut être modélisée grâce au module système suivant (exemple inspiré de [Clavel et al., 2007]).

```
mod SIMPLE-CLOCK is
 protecting INT .
 including MODULO .
 sort Clock .
 op clock : Int -> Clock .
 var T : Int .
 rl clock(T) => clock((T + 1) modu 24) .
endm
```

Avec un état initial `clock(0)`, grâce au calcul de modulo (`modu 24`), alors la valeur de l'horloge devrait être comprise uniquement entre 0 et 23. Soit `clock(T)` le pattern atteint, alors il devrait être tel que $0 \leq T < 24$. Cette propriété est vérifiable grâce à la commande suivante.

```
Maude> search in SIMPLE-CLOCK : clock(0) =>* clock(T)
                                    such that T < 0 or T >= 24 .
search in SIMPLE-CLOCK : clock(0) =>* clock(T)
                             such that T < 0 or T >= 24 = true .

No solution
```

Le `No solution` apparaissant à la fin de l'exécution signifie qu'il n'y a aucun élément de l'horloge inférieur à 0 ou supérieur ou égal à 24. La propriété est donc vérifiée. ◀

EXEMPLE *3.25*
Soit l'exemple 3.22 de l'algorithme du boulanger. Supposons que l'on souhaite vérifier une propriété d'*atteignabilité* telle que : "il arrive que le deuxième processus soit en section critique pendant que le premier processus attends". Ceci peut être modélisé par la commande suivante :

```
Maude> search [2] in BAKERY : initial =>* state(wait,N1:Nat, crit, N2:Nat) .
search in BAKERY : initial =>* state(wait, N1:Nat, crit, N2:Nat) .

Solution 1 (state 8)
states: 9  rewrites: 23 in 0ms cpu (0ms real) (38655 rewrites/second)
N1:Nat --> 2
N2:Nat --> 1
Solution 2 (state 15)
states: 16  rewrites: 44 in 1ms cpu (1ms real) (28478 rewrites/second)
N1:Nat --> 3
N2:Nat --> 2
```

Cette commande va permettre à `Maude` de renvoyer les deux premiers états du système vérifiant cette propriété (car la commande `search` est appelée avec le paramètre `[2]`). Notons que le `<terme 2>` n'est pas forcément un terme clos comme le `<terme 1>`, mais peut être un *pattern* contenant des variables (ici `N1` et `N2`).

Supposons que l'on souhaite vérifier une propriété de *sûreté* telle que : "deux processus n'entrent jamais en section critique au même moment". Ceci peut alors être modélisé par la commande suivante :

```
Maude> search in BAKERY : initial =>* state(crit,N:Nat, crit, N:Nat) .
search in BAKERY : initial =>* state(crit, N:Nat, crit, N:Nat) .
```

Comme nous l'avons vu, l'algorithme du boulanger est un système infini. Il n'est donc pas possible de vérifier une propriété de sûreté car cela signifie pouvoir calculer l'ensemble des états accessibles. Or ici cet ensemble n'est pas calculable et la commande ci-dessus boucle à l'infini. En entrant un paramètre limitant la profondeur de l'espace de recherche, on peut lancer cette commande et obtenir un résultat :

```
Maude> search [,1000] in BAKERY : initial =>* state(crit,N:Nat, crit, N:Nat) .
search [, 1000] in BAKERY : initial =>* state(crit, N:Nat, crit, N:Nat) .

No solution.
states: 2667  rewrites: 9327 in 27ms cpu (37ms real) (337567 rewrites/second)
```

La propriété de sûreté est vérifiée pour les 1000 premiers états de l'algorithme. ◀

Comment faire pour qu'une propriété de sûreté soit vérifiable en un temps fini pour tous les états accessibles du système (et pas seulement les 1000 premiers comme dans l'exemple ci-dessus), pour un système à états infinis tel que l'algorithme du boulanger ?

Dans [Meseguer et al., 2003], les auteurs proposent alors d'utiliser l'abstraction équationnelle, c'est-à-dire utiliser l'ensemble d'équations définissable dans un module en tant que fonctions d'abstraction. Comme nous l'avons vu dans les sous-sections précédentes, cet ensemble d'équations (qui ne sont en fait qu'un autre type de règles de réécriture) doit respecter plusieurs propriété (*Church-Rosser*, terminaison, et cohérence avec le système de réécriture, au moins pour les termes clos).

EXEMPLE *3.26*
Pour prouver la propriété de sûreté "deux processus n'entrent jamais en section critique au même moment" sur l'ensemble des états accessibles, il faut alors abstraire le calcul grâce à un ensemble d'équations. On obtient le module suivant, qui utilise les règles de réécriture du module `BAKERY` décrit dans l'exemple 3.22 (l'exemple de l'algorithme du boulanger et son abstraction sont définis dans [Meseguer et al., 2003]).

```
mod ABSTRACT-BAKERY is
 including BAKERY .

 vars P Q : Mode .
 vars X Y : Nat .

 eq state(P, 0, Q, s s Y) = state(P, 0, Q, s 0) .
 eq state(P, s s X, Q, 0) = state(P, s 0, Q, 0) .
 ceq state(P, s X, Q, s s Y) = state(P, s s 0, Q, s 0) if (s Y < X) .
```

```
 ceq state(P, s s s X, Q, s Y) = state(P, s s 0, Q, s 0) if (Y < s s X) .
 ceq state(P, s X, Q, s s Y) = state(P, s 0, Q, s 0) if not (s Y < X) .
 ceq state(P, s s X, Q, s Y) = state(P, s 0, Q, s 0) if not (Y < s X) .

endm
```

L'ensemble ci-dessus est *Church-Rosser*, terminant et cohérent avec le système de réécriture du module BAKERY.

Exécutons maintenant la commande search sur le module ABSTRACT-BAKERY afin de montrer la propriété de sûreté.

```
Maude> search in ABSTRACT-BAKERY : initial =>* state(crit,N1:Nat, crit, N2:Nat) .
search in ABSTRACT-BAKERY : initial =>* state(crit, N1:Nat, crit, N2:Nat) .

No solution.
states: 9  rewrites: 51 in 6ms cpu (56ms real) (7737 rewrites/second)
```

◀

L'outil Maude peut vérifier des propriétés d'atteignabilité, de sûreté, mais également des propriétés de vivacité. Nous venons de voir grâce à la commande search une manière de vérifier les deux premier types de propriétés. Mais ces propriétés peuvent aussi être vérifiée grâce à un module MODEL-CHECKER fournit dans Maude, capable également de vérifier des propriétés de vivacité ou d'autres propriétés temporelles de la logique LTL. Les propriétés d'atteignabilité et de sûreté peuvent être vues comme des invariants à vérifier, et pour pouvoir les vérifier grâce au model-checker Maude, ainsi que les propriétés de vivacité et autres propriétés LTL, il est nécessaire de définir un ensemble de prédicats sous forme d'équations. Ainsi, lors de la vérification de propriétés LTL, il y a deux ou trois niveaux de spécification :

- une spécification système fournie par un module système décrivant le comportement du système, avec règles de réécriture, et éventuellement équations modélisant les axiomes du système,
- éventuellement une spécification de l'abstraction fournie par un ensemble d'équations permettant de calculer une sur-approximation de l'ensemble des états accessibles si celui-ci est infini, et
- une spécification des prédicats utilisés pour vérifier des propriétés LTL sur le système modélisé.

Après importation du module MODEL-CHECKER, la syntaxe des prédicats doit être déclarée grâce à l'opérateur `op _|=_ : State Prop -> Bool .` contenu dans le module importé.

EXEMPLE *3.27*
Continuons avec l'exemple 3.22 du boulanger, contenant les équations d'abstractions vues dans l'exemple 3.26. Alors deux propriétés intéressantes à vérifier pourraient être :

(1) l'exclusion mutuelle, déjà vérifiée par le biais de la commande search dans l'exemple 3.26 : "deux processus n'entrent jamais simultanément en section critique".

(2) la vivacité : chaque processus en attente (état wait), va finir par entrer en section critique (état crit) un jour.

Pour vérifier ces propriétés sur l'algorithme du boulanger, il faut déclarer l'ensemble de prédicats suivants.

```
mod BAKERY-PREDS is
 protecting BAKERY .
 including MODEL-CHECKER .
 subsort BState < State .

 ops 1wait 2wait 1crit 2crit : -> Prop [ctor] .

 vars P Q : Mode .
 vars X Y : Nat .

 eq state(wait, X, Q, Y) |= 1wait = true .
 eq state(sleep, X, Q, Y) |= 1wait = false .
 eq state(crit, X, Q, Y) |= 1wait = false .
 eq state(P, X, wait, Y) |= 2wait = true .
 eq state(P, X, sleep, Y) |= 2wait = false .
 eq state(P, X, crit, Y) |= 2wait = false .
 eq state(wait, X, Q, Y) |= 1crit = false .
 eq state(sleep, X, Q, Y) |= 1crit = false .
 eq state(crit, X, Q, Y) |= 1crit = true .
 eq state(P, X, wait, Y) |= 2crit = false .
 eq state(P, X, sleep, Y) |= 2crit = false .
 eq state(P, X, crit, Y) |= 2crit = true .

endm
```

Puis le module contenant les trois spécifications détaillées plus haut (système, abstraction, prédicats) est le suivant (rappelons que le module ABSTRACT-BAKERY est détaillé dans l'exemple 3.26 :

```
mod ABSTRACT-BAKERY-PREDS is
 protecting ABSTRACT-BAKERY .
 including BAKERY-PREDS .
endm
```

◀

Une fois l'ensemble de prédicats construit, il faut exécuter le model-checker de `Maude` pour un terme initial donné et une formule LTL à vérifier, grâce à la commande suivante :

```
reduce modelCheck(<terme initial>,<formule LTL>) .
```

EXEMPLE *3.28*
Continuons l'exemple 3.27.

- La propriété (1) est alors modélisée par la formule LTL suivante : $\Box(\neg(1crit \wedge 2crit))$, et se traduit dans `Maude` par `[]~(1crit /\ 2crit))`.
- la propriété (2) est modélisée par la formule LTL : $(1wait \rightsquigarrow 1crit) \wedge (2wait \rightsquigarrow 2crit)$, et se traduit dans `Maude` par `(1wait |-> 1crit) /\ (2wait |-> 2crit)`.

Pour vérifier la propriété (1), on entre alors les commandes suivantes.

```
Maude> load model-checker.maude
Maude> load Bakery/bakery-preds.maude
Maude> load Bakery/abstract-bakery-preds.maude
Maude> reduce modelCheck(initial,[]~(1crit /\ 2crit)) .
reduce in ABSTRACT-BAKERY-PREDS : modelCheck(initial, []~ (1crit /\ 2crit)) .
rewrites: 70 in 1ms cpu (33ms real) (40183 rewrites/second)
result Bool: true
```

On arrive au résultat `true`, ce qui signifie que la propriété est bien vérifiée. Pour vérifier la propriété (2), on entre alors la commande suivante.

```
Maude> reduce modelCheck(initial,(1wait |-> 1crit) /\ (2wait |-> 2crit)) .
reduce in ABSTRACT-BAKERY-PREDS : modelCheck(initial, (1wait |-> 1crit) /\ (
    2wait |-> 2crit)) .
rewrites: 103 in 0ms cpu (1ms real) (299418 rewrites/second)
result Bool: true
```

On constate alors que la propriété est vérifiée. ◀

Dans le cadre de la vérification de propriété LTL grâce au model-checker, dans le cas où une abstraction est nécessaire, il faut alors que l'ensemble d'équations E modélisant l'abstraction satisfasse une autre propriété : préserver les prédicats que l'utilisateur souhaite vérifier (*invariant-preserving*). Pour chaque prédicat $pred$ à vérifier et pour tous termes clos t, t' tels que $t =_E t'$, alors E doit vérifier $pred(t) = pred(t')$. Autrement dit, les équations doivent préserver les booléens (*i.e.* le module `BOOL` prédéfini dans `Maude`). En effet, supposons t, t' deux termes ne vérifiant le même prédicat $pred$, tel que nous le voyons ci-dessous :

```
a |= pred = true .
b |= pred = false .
```

alors l'équation `eq a = b` ne préserve pas les prédicats, car celle-ci implique l'égalité `true = false`.

Une autre propriété doit être encore vérifiée dans le cas de prédicats et d'équations d'abstraction : le système de réécriture ne doit en effet pas posséder de *deadlocks* (*deadlock-free*), *i.e.* les termes doivent toujours pouvoir se réécrire. Par exemple, le système de réécriture $\{a \to b, b \to c\}$ possède un *deadlock* car il est bloqué une fois le terme réécrit en c : aucune règle ne permet de réécrire c, ce qui ne pose aucun problème en cas habituel, sauf quand des équations d'abstractions sont définies. Regardons le module `FOO` suivant, modélisant ce système de réécriture ainsi que quelques prédicats (exemple tiré de [Meseguer et al., 2003]).

```
mod FOO is
 including MODEL-CHECKER .
 ops a b c : -> State [ctor] .
 ops p1 p2 : -> Prop [ctor] .

 eq (a |= p1) = true .    eq (a |= p2) = false .
 eq (b |= p2) = true .    eq (b |= p1) = false .
 eq (c |= p1) = true .    eq (c |= p2) = false .
 rl a => b .              rl b => c .
endm
```

Alors la propriété $\Box(\Diamond(p2))$ (on a toujours éventuellement $p2$) n'est pas vérifiée par ce système : en effet, on a qu'une seule fois le terme b puis le calcul s'arrête. Ceci est confirmé par le model-checker de Maude.

```
Maude> reduce in FOO : modelCheck(a,[](<>(p2))) .
reduce in FOO : modelCheck(a, []<> p2) .
rewrites: 12 in 0ms cpu (312ms real) (23166 rewrites/second)
result ModelCheckResult: counterexample({a,unlabeled} {b,unlabeled}, {c,
    deadlock})
```

Jusqu'ici pas de problème. Rajoutons maintenant l'équation d'abstraction `eq c = a`. Notons que cette équation possède toute les propriétés requises : *Church-Rosser* terminante, cohérente avec l'ensemble des règles de réécriture énoncées dans le module ci-dessus, et conserve les prédicats (`a` et `c` vérifient les même prédicats). Le module représentant l'abstraction du système incluant cette équation d'abstraction, appelé `ABSTRACT-FOO`, est donc sensé représenter une sur-approximation des configurations accessibles. Malheureusement, ce module vérifie la propriété $\Box(\Diamond(p2))$ alors qu'elle n'est pas vérifiée par le module `FOO` dépourvu d'abstraction.

```
Maude> reduce in ABSTRACT-FOO : modelCheck(a,[](<>(p2))) .
reduce in ABSTRACT-FOO : modelCheck(a, []<> p2) .
rewrites: 12 in 0ms cpu (0ms real) (56338 rewrites/second)
result Bool: true
```

Ceci est dû au fait que le système de réécriture possède un *deadlock* en `c`. Une solution à ce problème est proposée dans [Meseguer et al., 2003]. Intuitivement, cette solution consiste à caractériser l'ensemble des états *deadlocks* en rajoutant un nouveau prédicat `enabled : Type -> Bool` pour chaque type `Type` du système de réécriture, et en rajoutant, pour chaque règle de réécriture `rl l=>r` du module, une équation `eq enabled(l)=true` (voir [Meseguer et al., 2003] pour plus de détails).

3.4.5 Comparaison Maude/Timbuk

L'outil `Maude` est un outil très générique et puissant, notamment d'un point de vue vérification. La classe de système de réécriture qu'ils utilisent pour modéliser un système est plus générique que pour la complétion d'automates d'arbres : en effet, il ne sont pas limités aux systèmes de réécriture (conditionnels) linéaires gauches. De plus, les équations peuvent également être conditionnelles, ce qui n'est pas le cas dans Timbuk. Cependant, l'utilisation d'un certain type de règles conditionnelles dans la complétion d'automates d'arbres a été implémentée durant cette thèse dans une extension de Timbuk appelée TimbukLTA (chapitre 6). `Maude` permet la vérification de propriétés d'atteignabilité et de sûreté, comme c'est le cas pour Timbuk. `Maude` définit également une abstraction équationnelle [Meseguer et al., 2003] quand le calcul est infini.

Mais la généricité de `Maude` entraine une forte contre-partie. En effet, les différentes spécifications utilisées pour modéliser un système, l'abstraire ou encore le vérifier doivent vérifier un nombre non négligeable de propriétés. Ainsi, nous avons vu dans les sous-sections précédentes que l'ensemble des équations E devait être *Church-Rosser* terminant, au moins pour les termes clos (*ground Church-Rosser terminating*). Que couplé à système de réécriture $\mathcal{R}$, l'ensemble E devait également être *cohérent* avec $\mathcal{R}$, au moins pour les termes clos (*ground coherent*).

Ces propriétés constituent une forte restriction de l'ensemble d'équations E utilisé en tant que fonction d'abstraction, et l'ensemble d'équations nécessaire pour abstraire certains système peut ne pas posséder ces propriétés. Ainsi, trouver un tel ensemble E n'est pas une tâche aisée. Comparons par exemple l'algorithme du boulanger dans `Maude` (exemples 3.22 et 3.26) et dans Timbuk(section 2.5 du chapitre 2). On constate que l'abstraction dans le cas de `Maude` nécessite six équations, sur lesquelles doivent être prouvées les différentes propriétés énumérées précédemment. Comme indiqué dans [Meseguer et al., 2003], afin de respecter toutes les propriétés, ces équations sont plus complexes qu'il n'est nécessaire pour abstraire le calcul. Dans Timbuk, il n'est pas possible d'avoir la même approximation. Cependant, bien qu'étant moins précise que celle définie dans `Maude`, une seule équation $s(s(s(s(x)))) = s(x)$ est nécessaire pour abstraire le calcul et prouver la propriété "deux processus n'entrent jamais dans la section critique au même moment". De plus, aucune preuve n'est demandée sur cette équation. La définition d'une approximation est de ce fait bien souvent plus facile et faisable dans Timbuk. En outre, si l'on reprend l'exemple de l'algorithme du boulanger, son implémentation sera similaire dans Timbuk quelque soit le nombre de processus : en effet pour n processus, l'équation d'abstraction sera alors $s^{n+2}(x) = s(x)$.

En se restreignant aux systèmes de réécritures linéaires gauches, la complétion d'automates d'arbres possède l'avantage de ne pas nécessiter de preuves d'autres propriétés pour garantir la sûreté du système, et ne possède pas d'autres restrictions sur son ensemble d'équations. Cependant, dans `Maude`, lorsque l'ensemble d'équations, respectant toutes les propriétés nécessaire, peut être défini, alors les propriétés de sûreté et les propriétés *temporelles* peuvent être vérifiées de manière très efficace. En effet, un des nombreux avantages de `Maude` est de pouvoir vérifier, grâce à son *model-checker*, les propriétés de vivacité et autres propriétés temporelles. Bien qu'il existe une méthode théorique permettant la vérification de propriétés temporelles par la complétion d'automates d'arbres (voir [Boyer et Genet, 2009] et section 3.2.2), elle n'a pas encore été implémentée dans Timbuk. En contre-partie, la vérification de propriétés temporelles dans `Maude` nécessite un ensemble de prédicats, qui implique, comme nous l'avons vu dans cette section, une nouvelle contrainte à respecter pour l'ensemble d'équations E du module : E doit également *préserver* les prédicats (*invariant-preserving*), et le système de réécriture $\mathcal{R}$ ne doit pas comporter de *deadlocks* (*deadlocks-free*).

Les automates d'arbres permettent de modéliser un ensemble infini de termes dans le cas de la complétion. C'est également le cas dans `Maude` grâce à un outil assez puissant amélioré récemment [Bae et al., 2013], appelé le `narrowing`. Cet outil permet une utilisation particulière de la commande `search`, où le terme initial peut contenir des variables libres. On peut par exemple avoir la commande `search s(0) + X  >* s(Y)` permettant de chercher l'ensemble des `X,Y` tels que `Y-1 = X+1`. Cependant, le *narrowing* ne fonctionne plus dès lors que des règles ou équations conditionnelles sont présentes dans la spécification.

De plus l'outil `Maude`, en cas d'approximation trop grossière, n'est pas adapté au raffinement, contrairement à la complétion d'automates d'arbres, comme nous l'avons vu en section 3.3.6 (outil TimbukCEGAR [Boichut et al., 2012]).

3.5 Application concrète à l'analyse de programme

Dans cette section nous allons donner un aperçu –nécessairement partiel– de différentes méthodes de vérification formelle, dans le cadre d'une application concrète à l'analyse de programmes tels que des programmes en C, en $Java$, ou encore des protocoles de sécurité.

Cet état de l'art, vu du côté "applicatif", va s'organiser selon deux différentes manières de vérifier un programme :

(1) **Programme** $\longrightarrow$ **Modèle Formel** $\overset{\text{Abstraction}}{\rightsquigarrow}$ **Vérification** : traduit exactement un programme en un modèle formel (comme par exemple un système de réécriture) afin de représenter la sémantique du programme, *i.e.* l'exact comportement du système. Puis le modèle généré est vérifié.

(2) **Programme** $\overset{\text{Abstraction}}{\rightsquigarrow}$ **Modèle Abstrait** $\longrightarrow$ **Vérification** : consiste à analyser le programme source et à en tirer un modèle abstrait adapté à ce que l'on souhaite vérifier, ne représentant qu'une abstraction des comportements du programme. Cette méthode a tout d'abord été introduite par l'*interprétation abstraite* [Cousot et Cousot, 1977] (section 2.6), qui en analysant le programme construit une sémantique abstraite de celui-ci.

La complétion d'automate d'arbres, dans le cadre de la vérification de programmes $Java$, se situe dans la catégorie (1), mais dans le cas de la vérification de protocoles de sécurité, c'est une abstraction du fonctionnement du protocole qui est vérifiée, et dans ce cas la complétion se place dans la catégorie (2).

3.5.1 Traduction exacte de la sémantique

La transformation du programme en un modèle formel afin d'effectuer la vérification sur ce dernier constitue un domaine vaste. Nous nous limiterons ici au cas où le modèle formel est un système de réécriture.

Différentes applications de la complétion d'automates d'arbres

Dans le cadre de la complétion d'automates d'arbres, plusieurs applications ont été mises en œuvre. Certaines de ces applications permettent une traduction exacte de la sémantique des programmes $Java$, comme c'est le cas dans [Boichut et al., 2007]. Le comportement de la $Java$ Virtual Machine sur le bytecode d'un programme peut en effet être modélisé par un système de réécriture. Afin d'implémenter cette modélisation, l'outil Copster [Barré et al., 2009] permet de transformer, de manière *automatique*, un programme $Java$ en un système de réécriture, dans la spécification utilisée par Timbuk, ainsi que dans celle de Maude. Le fichier généré peut ensuite être exécuté par un de ces deux outils, afin de permettre la vérification du programme. Mais une limite de ces travaux est l'utilisation des entiers de Peano pour représenter les entiers du programme, ce qui, dans certains cas, ralentit considérablement le calcul de complétion. Une amélioration convaincante sera alors proposé dans le chapitre 6, où nous décrirons plus en détail l'outil Copster.

Très récemment, la complétion d'automates d'arbres a été adaptée à la vérification de programmes fonctionnels tels qu'`Ocaml` [Genet et Salmon, 2013]. Ces travaux donnent une condition suffisante de terminaison de la complétion dans le cas d'une représentation d'un programme fonctionnel par un système de réécriture. Les auteurs enrichissent également l'algorithme de complétion avec des stratégies de réécriture, afin de prendre en compte la stratégie d'appel par valeur de nombreux langages fonctionnels.

Autres outils de réécriture et leurs applications

L'outil `Maude` vu en section précédente permet également de modéliser le fonctionnement d'un programme par un système de réécriture. Ainsi, un outil appelé `JavaFAN` [Farzan et al., 2004] a été implémenté dans le langage `Maude` pour la vérification des programmes $Java$. `JavaFAN` permet de modéliser des programmes $Java$ *multithreadé*, de détecter des violation de sûreté dans des espaces d'états infinis, et de vérifier des propriétés temporelles dans le cas *fini*. L'implémentation, prenant soit un programme $Java$, soit un bytecode $Java$ en entrée, est composée de 3000 lignes de codes `Maude`. La sémantique $Java$ représentée comprend le *multithread*, l'héritage, le polymorphisme, les références, et l'allocation dynamique d'objets. Cependant les méthodes natives et la majorité de librairies $Java$ ne sont pas prises en charge, contrairement à Copster. `JavaFAN` est un outil plutôt efficace, mais –contrairement à Copster et Timbuk– ne permet pas de vérifier de propriété de sûreté dans un espace d'états infini : en effet, il est capable de détecter une mauvaise configuration si elle existe, mais ne peut pas prouver qu'un terme n'est pas atteignable par le programme. Il n'implémente aucune abstraction.

À l'instar de `Maude`, un autre outil est également basé sur la logique de réécriture (*rewriting logic* [Bruni et Meseguer, 2003]). Il s'agit du $\mathbb{K}$-*framework* [Roşu, 2006, Rosu et Serbanuta, 2013], fournissant une notation permettant de définir la syntaxe et la sémantique d'un langage de programmation, ainsi qu'une série d'outils incluant un parseur et un interpréteur. Il permet aussi d'intégrer la gestion de la concurrence grâce à des règles de réécriture particulières étant plus concises et plus modulaires que les règles de réécriture standards. Il a déjà été utilisé pour définir la sémantique de programmes de langages courants tels que $Java$, `C` et `Scheme`. Les états (ou configurations) sont représentés par des structures imbriquées appelées cellules. Ces structures imbriquées peuvent être vues comme des termes spéciaux. La fonction de transition est quant à elle représentée par des règles de réécriture particulières permettant de réécrire ces cellules.

Ces cellules permettent de distinguer les différents composants d'un programme telles que les allocations mémoires, les piles de fonctions ou encore les multi-ensembles de *threads*. Une cellule représentant un *thread* peut contenir un ensemble d'autres cellules représentant par exemple son environnement ou son identifiant. Le $\mathbb{K}$-*framework* permet également de calculer des abstractions de programmes, comme par exemple une abstraction par prédicats [Asavoae et Asavoae, 2010], et a conduit à la conception de quelques outils d'analyse et de vérification [Regehr et al., 2012, Rosu et al., 2009, Roşu et Ştefănescu, 2011].

`Tom` [Tom, 2008] est un langage basé sur le calcul de réécriture permettant d'étendre des langages généralistes tels que `C`, $Java$, `Python`, `C++`, `Ada`, etc. Il enrichit ces langages de fonctionnalités issues du monde de la programmation fonctionnelle telles que le

pattern-matching (filtrage de motif) ou les stratégies de réécriture. D'un point de vue implémentation, Tom peut être vu comme un compilateur acceptant plusieurs langages hôtes, et dont le processus de compilation consiste à traduire les constructions de *matching* dans le langage hôte.

Tom manipule des structures arborescentes, ce qui en fait un excellent outil de manipulation de documents XML, de transformation de programmes grâce à la réécriture, ou encore, plus récemment, de transformation de modèles [Bach et al., 2012]. Il implémente également la complétion sous une autre forme, en utilisant des stratégies de réécriture. La complétion d'automates d'arbres telle que nous l'avons décrite dans cette thèse a également été implémentée en Tom [Balland et al., 2008]. Les transformations de langages implémentées par Tom permettent également la vérification de programmes.

En effet, des travaux axés sur la vérification de programmes $Java$ [Balland et al., 2007] permettent d'étendre Tom afin qu'il exprime des *transformations de bytecode*. Une transformation du bytecode d'un programme consiste par exemple à lui ajouter ou lui enlever des méthodes, des champs, ou des constructeurs, à changer les types ou les signatures des champs ou des méthodes, rediriger les invocations de méthodes vers de nouvelles cibles (pour appeler par exemple des méthodes sécurisées), et insérer du nouveau code dans les méthodes. L'approche de la transformation de bytecode peut aussi être utilisée pour spécifier des propriété de sécurité.

Basés sur des travaux de transformations de codes, notamment l'outil ASM [Bruneton et al., 2002], les travaux décrits dans [Balland et al., 2007] permettent de donner une implémentation en Tom d'un programme $Java$, permettant de manipuler et représenter les classes $Java$. Ces travaux permettent de donner une traduction précise de la sémantique $Java$ à l'instar de Copster dans le cas de la complétion, à la différence près qu'ils sont capables de vérifier des propriétés temporelles de la logique CTL. En effet, ces travaux permettent de démontrer que les stratégies de Tom permettent d'exprimer des propriétés CTL sur le code d'une méthode. Le mécanisme d'évaluation du langage de stratégies consiste alors en une procédure permettant de vérifier qu'une propriété CTL est vraie : une stratégie terminant sans faille assure la véracité de la propriété. Cependant, ces travaux constituent une première approche et ne sont pas optimisés. De plus, le traitement de programmes $Java$ possédant un espace potentiellement infini d'états accessibles n'est pas mentionné.

3.5.2 Construction d'un modèle abstrait du programme

Certaines techniques comportent une phase d'abstraction qui génère un modèle formel représentant une abstraction des comportements du système. Ces techniques sont généralement inspirées de l'*interprétation abstraite*.

Analyse statique par interprétation abstraite

L'analyse statique par interprétation abstraite [Cousot et Cousot, 1977] (détaillée en section 2.6) permet de vérifier un programme directement par analyse du code source, en calculant une sémantique abstraite du programme à partir de celui-ci, et les informations abstraites collectées peuvent être utilisées comme base de la vérification du programme.

L'analyse statique par interprétation abstraite est très efficace, et possède de nombreuses applications dans le milieu industriel. Cependant, la conception d'un l'analyseur

statique dépend souvent de la classe de programme ou du type de propriétés à vérifier, à partir desquels on va choisir la sémantique abstraite adaptée. Il n'y a donc pas la généricité que l'on peut retrouver dans le modèle de vérification vu en sous-section précédente. Des analyseurs statiques à but génériques existent, mais rencontrent alors des problèmes de temps d'exécution, et peuvent détecter un nombre non négligeable de faux contre-exemples, n'apportant ainsi pas d'avantages par rapport au modèle (1) décrit en sous-section précédente. Ainsi, les analyseurs statiques spécifiques, dédiés à un type de programme particulier et/ou une classe de propriétés à vérifier, sont très performants et détectent un pourcentage très faible de faux contre-exemples.

On pourra citer par exemple l'analyseur statique *Astrée* [Blanchet et al., 2002], dédié à l'analyse des logiciels embarqués temps-réel critiques, dont l'efficacité (moins d'une minute pour 10 000 lignes de codes) a permis la vérification de nombreux programmes critiques embarqués tels que le système de contrôle de vol de l'Airbus A360, un programme `C` de 132 000 lignes.

Il existe de nombreux autres analyseurs statiques par interprétations abstraites, tels que `Frama-C` [Cuoq et al., 2012], un *framework* plus général d'analyse et de preuve de code `C` implémenté en `Ocaml`. `Frama-C` permet également la vérification de programmes en `C`, `C++` ou `Ada`, mais concerne un ensemble restreint de propriétés à vérifier (division par zéro, dépassement arithmétique, valeurs potentiellement infinies, variables non initialisée, accès mémoire non valide, effets de bords dangereux). Cependant ce framework propose une extension des classes de propriétés vérifiables (comme par exemple les invariants) en utilisant la logique de *Hoare* [Hoare, 1969], mais cette extension est moins facile d'utilisation.

Enfin, pour la vérification de programmes $Java$, on peut également citer le $Java$ *Bytecode Verifier (BCV)*, qui permet de vérifier, par exemple, que la structure des fichiers `.class` est correcte, que les opérateurs reçoivent des arguments de type correct, ou qu'il n'y a pas d'objets non initialisés. Mais il ne permet pas de vérifier les booléens, les types génériques ou encore les interfaces.

Plus récemment, l'environnement `Sawja` [Hubert et al., 2010] est un outil d'analyse statique plus puissant permettant vérifier des programmes $Java$.

Pour résumer, l'analyse statique par interprétation abstraite est une méthode très performante, mais chaque analyseur statique est différent et très spécifique à une classe de programme et/ou de propriétés à vérifier. Une idée pourrait donc être d'utiliser l'efficacité de l'analyse statique par interprétation abstraite, mais la structure générique et formelle de la complétion d'automates d'arbres. Cette idée a été exploitée durant cette thèse et fait l'objet du chapitre 5.

Autres modèles abstraits

Nous avons vu dans la sous-section 3.5.1 des méthodes traduisant exactement la sémantique d'un programme. D'autres techniques existantes s'inspirent de l'interprétation abstraite pour construire un modèle représentant une abstraction du comportement du système.

Historiquement, une des premières applications de la complétion d'automate d'arbres a été la vérification de protocoles de sécurité [Genet et Klay, 2000]. Cette vérification représente une *abstraction* d'un protocole de sécurité par des règles de réécriture. La vérification d'un protocole de sécurité consiste, dans la majorité des cas, à vérifier qu'un

intrus ne peut pas s'emparer d'informations confidentielles échangées entre deux individus. Ainsi, chaque acteur du protocole (les deux individus souhaitant échanger des informations, ainsi que l'intrus malveillant) peut envoyer un message, crypté ou non, encodé par un terme. Par exemple, le terme $msg(x, y, c)$ représente le message envoyé de l'individu x à l'individu y, possédant le contenu c.

L'ensemble des messages qu'il est possible de construire par chaque individu dépend de sa *connaissance*, *i.e.* des informations qu'il a en sa possession, comme par exemple, la clé de déchiffrement, le mot de passe, etc. L'intrus peut envoyer des messages aux différents individus afin d'essayer d'enrichir sa connaissance. On doit alors vérifier qu'à aucun moment il ne peut obtenir dans sa connaissance une information confidentielle telle que la clé de déchiffrement. L'outil `AVISPA` [Armando et al., 2005], permettant de vérifier de nombreux protocoles, contient un outil de vérification basé sur la complétion d'automates d'arbres, appelé TA4SP [Boichut, 2005].

Les travaux de [Bouajjani et al., 2006b] permettent la vérification d'un sous-ensemble de programmes en `C`, grâce à la méthode de l'Abstract Regular Model-Checking [Bouajjani et al., 2006a] (vue en section 3.2.2 pour le calcul des successeurs et en sections 3.3.2 et 3.3.3 pour les fonctions d'abstractions). Rappelons que cette méthode utilise les transducteurs d'arbres comme fonction de transition et possède un algorithme d'abstraction-raffinement automatique (section 3.3.6).

La classe de programmes `C` vérifiée considère des programmes non-récursifs, dont les variables sont définies dans un domaine *fini* et manipulant des structures de données chaînées possédant possiblement plusieurs pointeurs sur l'élément suivant. Autrement dit, un élément de la structure chaînée peut posséder plusieurs voisins. Il peut s'agir par exemple de listes simplement ou doublement chaînées, mais aussi d'arbres ou de listes de listes. Les programmes manipulant ce genre de structures sont typiquement des systèmes à états infinis, et leurs configurations peuvent généralement être représentées par des graphes (que l'on appelle alors *shape graph* : graphe de forme). Les invariants sur la forme de ces graphes sont appelés *shape invariants*. Les propriétés vérifiées considérées dans ces travaux sont les suivantes :

- consistances des manipulations de pointeurs (aucun pointeur nul ne doit être assigné, aucun pointeur non-défini ne doit être utilisé, on ne doit pas faire référence à des éléments supprimés),
- pas de comportement non désiré : les *shape invariants* doivent être respectés par le programme.

Cette dernière propriété peut être vérifiée grâce à l'intégration, dans le programme source, de testeurs écrits en `C` modélisant l'erreur à ne pas atteindre. Ainsi, la vérification de ces propriétés est réduite à un problème d'atteignabilité.

EXEMPLE *3.29*
Le testeur `C` suivant permet de tester la consistance des pointeurs `next`, en étendant le langage `C` afin d'avoir la possibilité de remonter une opération de pointeur (l'opérateur `prev` du code ci-dessous).

```
while (x != NULL &&  random())
    x = x->next;
if (x != NULL && x->next->prev != x)
   error();
```

avec `x` la structure que l'on souhaite vérifier. Il suffit alors de vérifier que le code erreur `error()` n'est jamais atteint.

Les auteurs de [Bouajjani et al., 2006b] ont également créé une logique permettant de modéliser plus facilement les propriétés à vérifier. Cette logique, appelée LBMP (*Logic of Bad Memory Patterns*), n'est pas décidable, mais ses formules peuvent être traduites automatiquement sous forme de testeurs `C` et intégrées au programme source à vérifier. De la même manière, leur vérification est également réduite à un problème d'atteignabilité de la ligne de code correspondant à une erreur.

Afin de pouvoir vérifier ce problème d'atteignabilité grâce aux travaux de l'ARTMC [Bouajjani et al., 2006a], les ensembles de graphes de forme (*shape graphs*) du programme `C` représentant les configurations sont représentés par des automates d'arbres spéciaux, et les manipulations de pointeurs sont représentés par des transducteurs d'arbres possédant des extensions leur permettant de calculer les successeurs de ces automates d'arbres particuliers.

Nous allons maintenant expliquer brièvement cette représentation assez complexe. Tout d'abord, une structure en arbre est utilisée pour représenter le squelette d'un graphe de forme. Les arrêtes du graphe qui ne sont pas directement encodées dans ce squelette en arbre sont représentées par des expression de routage, *i.e.* des expressions régulières représentant les directions dans un arbre (telles que "en haut à gauche", "en bas à droite", etc.), et le type de nœud qui peut être visité lors du routage.

La figure 3.10 représente une traduction d'une liste doublement chainée sous forme d'arbre. Premièrement, il est à noter que les feuilles de l'arbres ne sont pas importante et pointent sur un symbole arbitraire (ici $\bullet$). Ensuite, chaque nœud de l'arbre est encodé d'une manière particulière :

(1) Le premier élément contient l'ensemble des **variables** pointant sur référence du nœud. Ici, X et Z sont des variables pointant sur le premier et le troisième nœud de la liste.

(2) Ensuite, on trouve les **marqueurs** du nœuds, *i.e.* par quel autre nœud ce nœud a été marqué pour être son successeur. Ici, dans le cas d'une liste doublement chainée, il s'agit forcément du nœud suivant et du nœud précédent, sauf pour les extrémités de la liste qui ne sont pointées que par un seul autre nœud.

(3) Puis le troisième élément du nœud contient la **valeur** du nœud (ici les entiers 7,14,26 et 10).

(4) Tous les éléments suivants du nœud correspondent aux différents sélecteurs de pointeurs de nœud suivant. Dans le cas d'une liste doublement chainée, chaque nœud possède deux successeurs, donc deux sélecteurs de pointeurs. Le nœud possède alors deux éléments pour représenter ses sélecteurs. Ces éléments contiennent les expressions de routage évoquées précédemment, indiquant donc vers quel successeur le nœud doit pointer. Dans le cas de la liste doublement chainée, chaque nœud pointe sur le nœud suivant et le nœud précédent, sauf pour les extrémités de la liste dont un des deux pointeurs pointe sur `null`. La figure 3.10 présente une version simplifiée des expressions de routage. Par exemple, "s_1 : nœud père" signifie que le premier sélecteur pointe sur le nœud père dans l'arbre. Les expressions de routage prennent également en compte les marqueurs des autre nœuds, mais ceci n'apparait pas sur la figure par soucis de simplification.

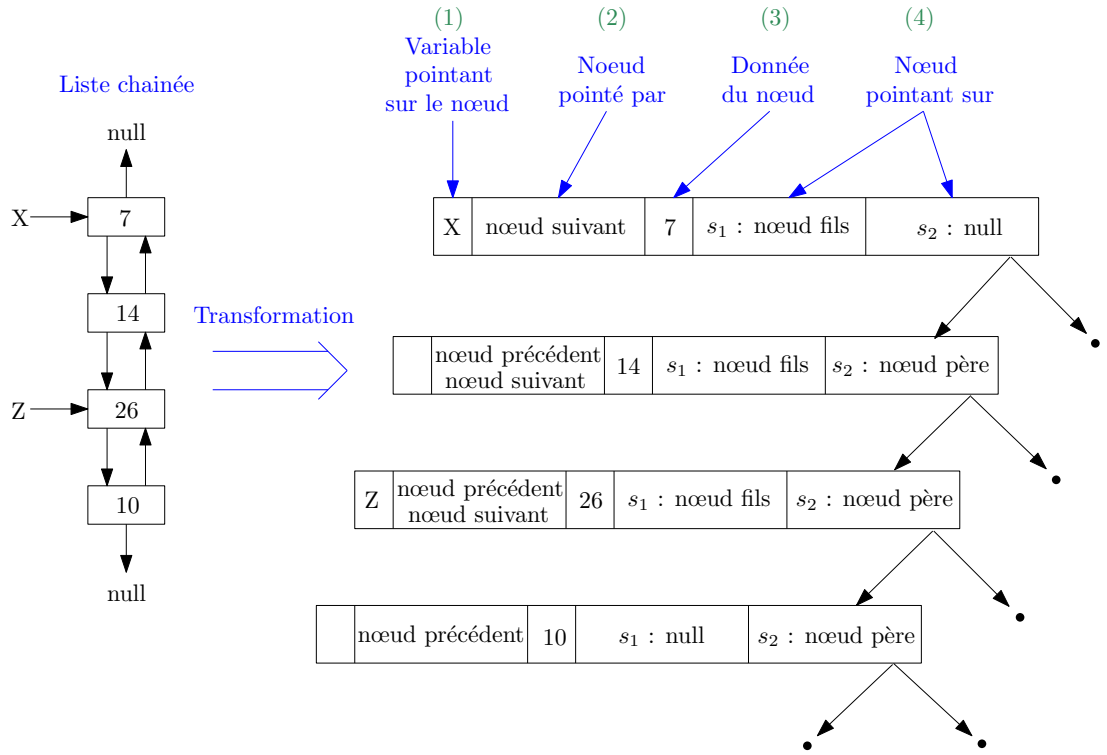


FIGURE 3.10 — Modélisation d'une liste doublement chainée par un arbre contenant des expressions de routage (figure inspirée de [Bouajjani et al., 2006b]).

Les transducteurs d'arbres permettant de modéliser les instructions de manipulation de pointeurs sont étendus afin de calcul les successeurs de ces automates d'arbres particuliers. Les auteurs utilisent également d'autres transducteurs dédiés à l'application des expressions de routage. Ce travail a été implémenté en utilisant l'outil `Mona` [Henriksen et al., 1995] et les expérimentations donnent des résultats concluants. De plus, la construction des automates d'arbres, des transducteurs et de certains testeurs est automatique, ainsi que la vérification.

Cependant, la modélisation ne représente qu'une abstraction du programme `C`, et cette méthode ne gère pas les programmes récursifs ni les domaines infinis, contrairement à l'outil Copster. En revanche, elle est dédiée à la vérification de propriétés sur la structure des données et des pointeurs, et la vérification de ce type de propriété n'est pour l'instant pas faisable avec Copster et Timbuk.

D'autres outils tels que `SLAM` et `BLAST` construisent également des modèles abstraits du programme, et la construction du modèle se fait **automatiquement** depuis le programme source. Ces outils utilisent un ensemble de prédicats pour transformer le programme à vérifier en un programme simplifié, plus facilement vérifiable. Le paramétrage de l'abstraction par des prédicats permet de construire des abstractions intéressantes et vérifiables de façon efficace, ce qui n'est pas sans rappeler l'abstraction par prédicat que nous avons vue en section 3.3.3.

`SLAM` [Ball et Rajamani, 2001] est un ensemble d'outils permettant de vérifier automatiquement des propriétés de sûreté sur des programmes en `C`. La vérification d'un programme P dans `SLAM` fonctionne alors de la manière suivante. Tout d'abord, les mauvaises configurations que l'on ne veut pas atteindre sont écrites sous forme de *prédicats*

dans un langage de spécification appelé SLIC. Puis ces prédicats sont ensuite compilés sous forme d'un programme C que l'on va noter Bad.

Le programme Bad des mauvaises configurations est alors inséré dans le programme P à vérifier, pour ensuite abstraire le programme résultat en un *programme booléen* P' grâce à l'outil C2BP. Un *programme booléen* est un programme C dans lequel le seul type est le type des booléens. L'intérêt d'une telle abstraction se situe principalement dans le fait que l'analyse d'atteignabilité et la terminaison, généralement indécidables, sont décidables dans le cas particuliers des programmes booléens. De plus, ils permettent de conserver la structure de flot de contrôle du programme d'origine, et sont donc particulièrement indiqués pour la vérification de propriétés temporelles. Cette abstraction de P relativement à Bad constitue alors une abstraction par prédicat,.

Ensuite, l'abstraction P' est vérifié grâce au model-checker BEBOP dédié aux programmes booléens, utilisant une modélisation sous forme de BDD (*Binary Decision Diagrams*). BEBOP [Ball et Rajamani, 2000] est un outil très efficace permettant de vérifier des milliers de lignes de codes et des centaines de procédures en quelques minutes, et dans le cas où une mauvaise configuration est atteinte, renvoie la trace qui permet de l'atteindre. Le fonctionnement de SLAM peut être schématisé de la manière suivante :

$$\text{Programme } P \overset{\text{Abstraction}}{\rightsquigarrow} \text{Programme booléen } P' \longrightarrow \text{Modèle} \longrightarrow \text{Vérification}$$

SLAM a notamment permis la vérification des *drivers* du *Microsoft Driver Development Kit*. C'est un outil automatique définissant une abstraction correcte et précise, et contenant également une procédure de raffinement. Cependant, la classe de propriétés qu'il peut vérifier est limitée. Il ne peut par exemple pas faire de vérification sur les valeurs numériques, de par l'abstraction booléenne.

Basé sur le même principe d'abstraction par prédicats, BLAST [Henzinger et al., 2003] permet également de vérifier automatiquement des programmes C, avec une procédure de raffinement. Les programmes C sont représentés par des automates de flots de contrôle (*CFA : Control Flow Automata*), qui sont des graphes de flots de contrôle dont les arêtes sont étiquetées par des opérateurs. BLAST vérifie un programme en construisant un arbre représentant une partie du *CFA* du programme à vérifier, et dont les nœuds sont abstraits par des prédicats d'abstraction. Par ces nombreuses heuristiques, BLAST constitue un outil plutôt efficace, mais les prédicats trouvés automatiquement ne sont parfois pas suffisants et la vérification nécessite l'ajout d'un ensemble de prédicats par l'utilisateur.

3.6 Conclusion et ouverture sur les contributions

La représentation par automates d'arbres et systèmes de réécriture, agrémentée de l'algorithme de complétion, est une méthode *générique* de calcul des configurations accessibles d'un système, permettant de modéliser de manière relativement simple de nombreux systèmes. Nous avons par exemple vu en section 3.2.2 les avantages et les inconvénients de cette représentation par rapport aux transducteurs d'arbres. Bien sur, dans le cas de systèmes à états infinis, la question de l'accessibilité d'une configuration est indécidable. Cependant, des méthodes d'abstractions sont proposées pour palier à ce problème.

La fonction d'abstraction choisie dans cette thèse est l'abstraction équationnelle, et elle permet d'exprimer de nombreuses abstraction sans aucune contrainte sur les équations, contrairement à ce qu'il en est dans l'outil `Maude` (section 3.4). Mais une de ses limites est que les équations doivent être définies à la main.

En contrepartie, cette méthode est agrémentée :

- D'une certification `Coq` : la complétion d'automates d'arbres avec abstraction équationnelle a été prouvée correcte en `Coq` [Boyer et al., 2008].
- D'un outil de raffinement automatique de l'abstraction : TimbukCEGAR [Boichut et al., 2012].

De plus, la terminaison de cette méthode a également été prouvée récemment pour certaines classe d'équations, respectant une certaine condition [Genet, 2014].

La complétion d'automates d'arbres semble ainsi être un bon compromis entre expressivité, précision, décidabilité et facilité d'utilisation. Cependant, des problèmes persistent, et nous avons dans cette thèse tenté d'améliorer deux d'entre eux.

I. Le premier problème se situe au niveau de l'abstraction : en effet, la définition des équations d'abstractions est manuelle. De ce fait, elle nécessite une certaine expertise de l'utilisateur, aussi bien sur le type de programme qu'il veut vérifier que sur la technique de complétion en elle-même. L'idée de la première partie des contributions est donc de caractériser, par des formules logiques et de manière *automatique*, ce qu'est une "bonne" sur-approximation : une approximation représentant un surensemble des configurations accessibles (donc correcte), et qui soit suffisamment précise pour ne pas reconnaitre de faux contre-exemples.

II. Le second problème est le passage à l'échelle, *i.e.* le temps de calcul parfois élevé du calcul de complétion quand on s'attaque à des problèmes de la vie courante. Dans les travaux sur la vérification de programmes $Java$ [Boichut et al., 2006], l'explosion du calcul de complétion peut notamment être due au traitement des entiers, qui sont représentés sous formes de termes par des entiers de Peano, et des opérations arithmétiques, qui sont représentées par des règles de réécriture. Ainsi l'exécution d'une opération arithmétique peut nécessiter l'application de centaines de règles de réécriture. Des équations d'abstractions peuvent être ajoutées sur ces entiers de Peano, mais alors on perd en précision concernant les informations sur les valeurs numériques prises par les différentes variables, et le nombre de propriétés que l'on peut vérifier s'en retrouve réduit.

    En section 3.5 nous avons mentionné l'efficacité et les nombreuses applications de l'interprétation abstraite. Cependant, chaque analyse statique nécessite d'être redéfinie selon la classe de programmes ou de propriétés que l'on souhaite vérifier. Afin de trouver un compromis entre la généricité de la complétion d'automate d'arbres et l'efficacité de l'interprétation abstraite, nous proposons donc un mélange de ces deux techniques, où le domaine abstrait correspondant le mieux au type de programme ou de propriété à vérifier pourrait être branché dans le modèle générique automate d'arbres/système de réécriture.

    En deuxième partie de cette thèse, nous avons donc adapté à notre problème l'idée décrite dans [Le Gall et Jeannet, 2007], consistant à intégrer des éléments d'un treillis abstrait dans un automate, appelé alors automate (de mots) à treillis.

Première partie

Automates d'arbres et formules logiques

Caractérisation de points-fixes concluants par formules logiques

4

Sommaire

4.1 Introduction **105**
4.2 Préliminaires **108**
4.3 Automates d'arbres symboliques **109**
4.4 Solutions de l'algorithme de filtrage pour les STA **114**
4.5 À la recherche d'un automate point-fixe concluant **121**
4.5.1 Formule point-fixe 122
4.5.2 Formule de non-reconnaissance des termes interdits 127
4.5.3 Caractérisation d'un point-fixe concluant 129
4.6 Analyse d'atteignabilité via résolution de formules **131**
4.7 Conclusion et perspectives **137**

"Faites que le rêve dévore votre vie,
afin que la vie ne dévore pas votre rêve."

Antoine de Saint Exupéry

Résumé

Dans la complétion d'automates d'arbres, la vérification d'une propriété de sûreté consiste à montrer qu'un ensemble de termes interdits Bad ne peut pas être atteint depuis un ensemble d'états initiaux. Étant donné que l'ensemble des états accessibles n'est généralement pas calculable, une sur-approximation de cet ensemble est généralement calculée. La principale difficulté de ce calcul est de calculer une sur-approximation suffisamment précise pour ne pas accepter de termes interdits, et apporter une réponse au problème de vérification. Autrement dit, cette sur-approximation doit être *concluante*. Comme nous l'avons vu en section 3.3, cette précision est le plus souvent définie *manuellement* par un paramétrage implémentant le calcul de la sur-approximation.

Nous proposons alors dans ce chapitre une caractérisation de ce qu'est une *sur-approximation concluante*, par des formules logiques générées grâce à un nouveau type d'automate appelé *automate d'arbres symbolique*. Résoudre ces formules, *pour une taille d'automate d'arbres symbolique donnée*, permet alors de trouver une sur-approximation concluante de *taille équivalente*, sans avoir recours à aucun paramétrage manuel. Si aucune solution n'est trouvée pour la taille donnée, alors la taille de l'automate d'arbres symbolique est augmentée et le calcul des formules est répété.

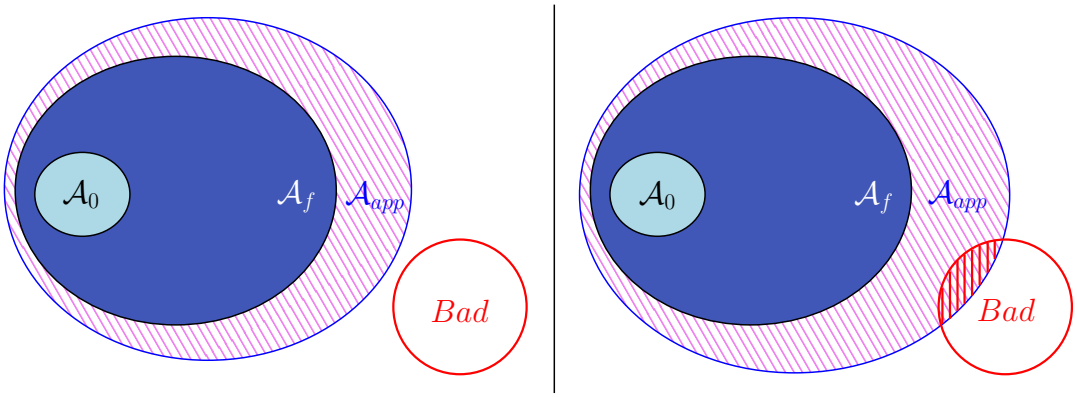


FIGURE 4.1 — Cas 1 : l'intersection entre la sur-approximation et l'ensemble des termes interdits est vide, la propriété est donc vérifiée. Cas 2 : la sur-approximation est trop grossière, on a un *faux contre-exemple*.

4.1 Introduction

Nous avons vu lors des chapitres précédents de cette thèse, que la technique de vérification que nous utilisons représente les ensembles de configurations du système par des automate d'arbres, et le comportement du système par un système de réécriture. Rappelons brièvement que dans ce contexte, les auteurs de [Genet et Rusu, 2010, Feuillade et al., 2004] proposent une analyse d'atteignabilité grâce à un algorithme de complétion (section 2.4). Soit Bad un ensemble de termes interdits, la complétion consiste donc à prouver qu'aucun des éléments de Bad n'est accessible, en calculant une sur-approximation régulière de l'ensemble des termes accessibles qui ne contienne aucun termes interdit. Cette technique est paramétrée par une fonction d'approximation ou un ensemble d'équations (section 2.4.2) : ce paramétrage détermine la qualité de l'approximation et se révèle donc crucial pour l'analyse d'atteignabilité.

En effet, nous pouvons voir sur la figure 4.1 que l'absence de termes interdits dans le langage de sur-approximation calculée par complétion ($\mathcal{A}_{app}$) garantit la sûreté du système. En revanche, si le langage de la sur-approximation calculée contient des termes interdits, il s'agit peut-être de faux contre-exemples. Dans ce cas, cela signifie que le paramétrage choisi pour l'approximation n'est pas le bon et que l'algorithme de complétion calcule une sur-approximation trop grossière. Afin d'éliminer les faux contre-exemples, on doit alors procéder à un raffinement de l'abstraction, mais cette procédure est généralement coûteuse (voir section 3.3.6).

Or il est difficile d'établir une bonne fonction d'approximation permettant de calculer une sur-approximation suffisamment fine pour de pas admettre de faux contre-exemple. Ce paramétrage requiert souvent une expertise hautement spécialisée pour pouvoir espérer une analyse concluante : le vérificateur doit alors maitriser à la fois la technique d'approximation utilisée et le type de système vérifié. Dans [Boichut et al., 2006], où les auteurs proposent une vérification des programmes Java par complétion d'automate d'arbres, les approximations doivent être très précisément définies pour permettre la convergence du système, nécessitant une étude préalable très pointue.

Nous proposons dans ce chapitre une technique permettant de calculer de manière automatique une *sur-approximation concluante*, afin de retirer tout paramétrage manuel

de la part de l'utilisateur. Une *sur-approximation concluante* est une approximation :

1. contenant tous les termes accessibles du système : elle est alors représentée par un automate *point-fixe* (voir section 2.4), et
2. ne reconnaissant aucun faux contre-exemple.

Par la suite, une *sur-approximation concluante* peut également être appelée *point-fixe concluant*. Afin de calculer un *point-fixe concluant*, nous caractérisons, par une formule logique, les critères permettant d'obtenir une telle approximation dans le cadre de la technique de complétion proposée par [Genet, 1998, Feuillade et al., 2004, Genet et Rusu, 2010]. Au lieu de raisonner avec des automate d'arbres classiques, ces derniers sont généralisés par un *automate d'arbres symbolique* $\mathcal{A}_S$ (*Symbolic Tree Automaton* : STA) où les états sont remplacés par des variables. Chaque variable X représentant un état d'un automate symbolique $\mathcal{A}_S$ peut être instanciée par un état "classique" q d'un automate d'arbres $\mathcal{A}$. Ainsi, une instance d'un STA est un automate d'arbres.

Soient $\mathcal{A}$ un automate d'arbres représentant l'ensemble des configurations initiales, et $\mathcal{R}$ un système de réécriture. Les formules caractérisant un point-fixe concluant pour $\mathcal{A}$ et $\mathcal{R}$ sont composées de combinaisons booléennes d'égalités ou d'inégalités sur les variables de l'*automate d'arbres symbolique* (STA) correspondant à $\mathcal{A}$. Étant donné que chaque instanciation d'un STA produit un automate d'arbres, une instanciation *valide* (*i.e.* satisfaisant les formules) d'un STA produit donc un automate d'arbres $\mathcal{A}'$ représentant une *sur-approximation concluante* de l'ensemble des termes accessibles depuis $\mathcal{A}$, selon la relation de réécriture induite par $\mathcal{R}$. De plus, $\mathcal{A}'$ ne reconnaitra aucun terme interdit.

En d'autres mots, dans le cas de techniques d'approximation "classiques", le successeur calculé à chaque étape de complétion est déjà déterminé par la fonction d'approximation ou l'ensemble d'équations. Il n'y a alors qu'un successeur possible, qu'un chemin possible, et un seul résultat. Si l'approximation n'est pas assez fine, il faut alors revenir en arrière et changer le paramétrage (voir section 3.3.6 et [Boichut et al., 2008a, Boichut et al., 2012]). Dans notre cas, un STA permet de représenter *tous* les successeurs possibles à une étape donnée de la complétion : toutes les instances possibles d'un STA représentent toutes les approximations possibles à une étape donnée. La figure 4.2 représente trois approximations possibles, selon trois instanciations différentes des variables X_1 à X_4 de l'automate d'arbres symbolique. Soit $\mathcal{A}_S$ le STA possédant les transitions suivantes : $\{a \to X_1, b \to X_2, f(X_1, X_2) \to X_3), f(X_3) \to X_4\}$, avec $X_1, \ldots, X_4$ des variables. Ces variables peuvent être instanciées par des états "classiques" (par exemple, les états q_1 à q_4) : l'ensemble de transitions du STA représente donc toutes les instances possibles de cet ensemble de transitions symbolique. Différentes instances possibles peuvent être : $\{a \to q_1, b \to q_2, f(q_1, q_2) \to q_3), f(q_3) \to q_3\}$, ou encore : $\{a \to q_1, b \to q_1, f(q_1, q_1) \to q_2), f(q_2) \to q_3\}$.

Le respect ou non des formules caractérisant un point-fixe concluant par une instanciation permet alors de déterminer quelle approximation, parmi toutes les approximations possibles à une étape donnée, est un point-fixe concluant. Trouver une sur-approximation concluante se réduit alors à une résolution de formules logiques, où différentes techniques de résolution et de recherche (issues, par exemple, de l'intelligence artificielle), peuvent être appliquées.

Ce chapitre est organisé comme suit : la section 4.2 permet de décrire le type de formule manipulée dans le reste du chapitre, ainsi que la notion d'instanciation correcte. La

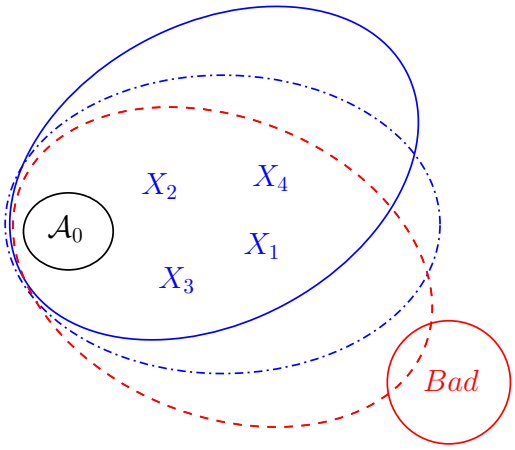


FIGURE 4.2 — L'ensemble des successeurs possibles représenté par un STA.

section 4.3 introduit et définit les automate d'arbres symboliques (STA), et présente également la connexion entre un STA et un automate d'arbres classique, basée sur les instanciations des variables du STA. Puis la section 4.4 décrit un nouvel algorithme de filtrage adapté aux STA. Pour cela, la condition d'existence d'un terme t donné dans le langage d'un STA est caractérisé par une formule logique. Ensuite, la section 4.5 présente notre principale contribution : la caractérisation d'une sur-approximation concluante par une formule logique. Finalement, la section 4.6 conclut ce chapitre par quelques expérimentations sur notre approche, grâce à l'implémentation d'un algorithme de recherche de point-fixe concluant en utilisant l'outil *Mona* [Henriksen et al., 1995]. Il est à noter que deux exemples fils rouges seront utilisés tout au long de ce chapitre. Le premier, très simple, permettra de comprendre rapidement les notions définies, et le second sera plus complexe afin d'illustrer au maximum les cas d'utilisation de nos algorithmes.

Travaux liés existants

Comme expliqué précédemment, les techniques liées à la complétion d'automates d'arbres utilisent une fonction d'approximation nécessitant un paramétrage manuel et une certaine expertise. Ce n'est pas le cas dans ce chapitre, puisque un automate représentant une sur-approximation des accessibles est trouvé de manière automatique.

Dans [Genet et Rusu, 2010], un ensemble d'équations doit être défini à la main. Afin que l'approximation point-fixe générée n'admettent aucun faux contre-exemple, cet ensemble d'équations doit être défini avec minutie et précision. Dans le cas contraire, un raffinement sera nécessaire et augmentera de manière non-négligeable la durée de l'analyse. Il en est de même pour l'abstraction par prédicat utilisée dans [Bouajjani et al., 2006a], où une expertise dans cette technique se révèle nécessaire.

Dans [Gallagher et Rosendahl, 2008], les auteurs encodent la complétion d'automates d'arbres par des programmes logiques sous forme de clauses de Horn. Par conséquent, ils utilisent de nombreux résultats obtenus dans le domaine de l'analyse statique de programmes logiques : ils sont alors capables de calculer des approximations précises dans le cadre de l'analyse statique.

Dans le domaine de la complétion d'automates d'arbres, une approximation est suffisamment précise si elle permet de montrer qu'un terme ou un ensemble de termes n'est

pas atteignable. Dans ce chapitre, l'approximation trouvée est nécessairement précise, puisqu'elle doit vérifier une contrainte définie à partir de l'ensemble des termes interdits. Une approximation vérifiant cette contrainte est donc nécessairement concluante, i.e. ne contient pas de terme interdit, ce qui n'est pas le cas des approximations trouvées par [Gallagher et Rosendahl, 2008]. De plus, le calcul de point-fixe de notre approche est orienté par un but : le respect des contraintes caractérisant un point-fixe concluant, alors que la plupart des approches de calcul de point-fixe, telles que celle définie dans [Gallagher et Rosendahl, 2008], vérifient l'accessibilité d'un terme seulement après calcul des états accessibles.

4.2 Préliminaires

Dans les définitions suivantes, nous introduisons les formules logiques que nous allons manipuler dans ce chapitre, ainsi que les notions d'instance et de satisfaction de ces formules.

DÉFINITION 4.2.1 ($\rho(\mathcal{X}_\mathcal{Q})$)
Soit $\mathcal{X}_\mathcal{Q}$ un ensemble de variables. On note ici $\rho(\mathcal{X}_\mathcal{Q})$ un ensemble de formules logiques sur $\mathcal{X}_\mathcal{Q}$ de la forme suivante :
- $\top, \bot \in \rho(\mathcal{X}_\mathcal{Q})$,
- $X = Y$, $X \neq Y \in \rho(\mathcal{X}_\mathcal{Q})$ *avec* $X, Y \in \mathcal{X}_\mathcal{Q}$, *et*
- $\varphi_1 \wedge \varphi_2$, $\varphi_1 \vee \varphi_2$, $\neg\varphi_1$, *et* $\varphi_1 \Rightarrow \varphi_2$ *appartiennent à* $\rho(\mathcal{X}_\mathcal{Q})$, *avec* $\varphi_1, \varphi_2 \in \rho(\mathcal{X}_\mathcal{Q})$.

EXEMPLE *4.1*
Soit $\mathcal{X}_\mathcal{Q}$ un ensemble de variables tel que $\mathcal{X}_\mathcal{Q} = \{X_1, X_2, X_3\}$. Alors $(X_1 \neq X_2) \wedge ((X_1 = X_3) \vee (X_2 = X_3))$ est une formule de $\rho(\mathcal{X}_\mathcal{Q})$. ◀

Comme nous le verrons dans la section suivante, les variables de $\mathcal{X}_\mathcal{Q}$ seront instanciées par les états d'un ensemble de symboles $\mathcal{Q}$. Une formule logique telle que définie ci-dessus est alors satisfaite pour une instanciation particulière des variables de $\mathcal{X}_\mathcal{Q}$ par des états de $\mathcal{Q}$. Définissons la notion d'instance et de satisfaction de nos formules logiques dans ce contexte.

DÉFINITION 4.2.2 (Instanciation/satisfaction)
Soit $\mathcal{Q}$ un domaine non-vide. Une instanciation $\mathcal{I}$ des variables de $\mathcal{X}_\mathcal{Q}$ est une fonction $\mathcal{I} : \mathcal{X}_\mathcal{Q} \to \mathcal{Q}$. L'instanciation $\mathcal{I}$ satisfait une formule $\varphi \in \rho(\mathcal{X}_\mathcal{Q})$ (noté $\mathcal{I} \models \varphi$), si et seulement si :

(i) $\mathcal{I} \models \top$,

(ii) $\mathcal{I} \models X_1 = X_2 \Leftrightarrow \mathcal{I}(X_1) = \mathcal{I}(X_2)$,
$\mathcal{I} \models X_1 \neq X_2 \Leftrightarrow \mathcal{I}(X_1) \neq \mathcal{I}(X_2)$,

(iii) $\mathcal{I} \models \varphi_1 \wedge \varphi_2 \Leftrightarrow \mathcal{I} \models \varphi_1$ *et* $\mathcal{I} \models \varphi_2$,
$\mathcal{I} \models \varphi_1 \vee \varphi_2 \Leftrightarrow \mathcal{I} \models \varphi_1$ *ou* $\mathcal{I} \models \varphi_2$,

(iv) $\mathcal{I} \models \neg\varphi_1 \Leftrightarrow \mathcal{I} \not\models \varphi_1$
$\mathcal{I} \models \varphi_1 \Rightarrow \varphi_2 \Leftrightarrow \mathcal{I} \not\models \varphi_1$ *ou* $\mathcal{I} \models \varphi_1 \wedge \varphi_2$,

avec $X_1, X_2 \in \mathcal{X}_\mathcal{Q}$ *et* $\varphi_1, \varphi_2 \in \rho(\mathcal{X}_\mathcal{Q})$

EXEMPLE *4.2*
Soit $D = \{1, 2\}$ et $\mathcal{I}$ l'instanciation telle que $\mathcal{I}(X_1) = 2$, $\mathcal{I}(X_2) = \mathcal{I}(X_3) = 1$. Alors $\mathcal{I}$ vérifie $\mathcal{I} \not\models X_1 = X_2$ et $\mathcal{I} \models (X_1 = X_2) \vee (X_2 = X_3)$. ◀

Notons que les instanciations seront également considérées en tant que substitutions dans le reste de ce chapitre, puisqu'étant des fonctions $\mathcal{I} : \mathcal{X}_\mathcal{Q} \mapsto \mathcal{Q}$. Une instanciation $\mathcal{I}$ s'applique sur un terme t (noté $t\mathcal{I}$) de la même manière qu'une substitution σ définie dans le chapitre 2 (définition 2.1.8).

4.3 Automates d'arbres symboliques (Symbolic Tree Automata : STA)

Dans cette section nous allons définir un nouvel outil théorique permettant de caractériser par des formules logiques une approximation point-fixe n'admettant pas de termes interdits : les automates d'arbres symboliques.

Soit $\mathcal{X}_\mathcal{Q}$ un ensemble de variables que nous appelons *états symboliques*. Un *automate d'arbres symbolique* (ou *Symbolic Tree Automata*, noté STA par la suite pour plus de simplicité) est un automate d'arbres dont les états sont des variables de $\mathcal{X}_\mathcal{Q}$, qui peuvent donc avoir différentes instanciations. Un STA est composé de *transitions symboliques normalisées* définies ci-dessous.

DÉFINITION 4.3.1 (Transition symbolique normalisée)
Soit $\mathcal{X}_\mathcal{Q}$ un ensemble d'états symboliques (ou variables). Une transition symbolique normalisée est de la forme suivante : $f(X_1, .., X_n) \to X$, avec $f \in \mathcal{F}^n$ et $X, X_1, .., X_n \in \mathcal{X}_\mathcal{Q}$.

Nous donnons maintenant la définition formelle d'un STA.

DÉFINITION 4.3.2 (Automate d'arbres symbolique (STA))
Un STA est un tuple $\langle \mathcal{F}, \mathcal{X}_\mathcal{Q}, \mathcal{X}^f_\mathcal{Q}, \Delta \rangle$, avec $\mathcal{X}_\mathcal{Q}$ un ensemble d'états symboliques, $\mathcal{F}$ un ensemble de symboles fonctionnels, $\mathcal{X}^f_\mathcal{Q} \subseteq \mathcal{X}_\mathcal{Q}$ un ensemble d'états finaux symboliques, et Δ un ensemble de transitions symboliques normalisées.

EXEMPLE *4.3*
Soit $\mathcal{A}_{S_1} = \langle \mathcal{F}, \mathcal{X}_\mathcal{Q}, \mathcal{X}^f_\mathcal{Q}, \Delta \rangle$ un STA, avec $\mathcal{F}$ un ensemble de symboles fonctionnels tel que $\mathcal{F} = \{a_0, s_1\}$, $\mathcal{X}_\mathcal{Q}$ et $\mathcal{X}^f_\mathcal{Q}$ deux ensembles d'états symboliques tels que $\mathcal{X}_\mathcal{Q} = \{X_0, X_1\}$ et $\mathcal{X}^f_\mathcal{Q} = \{X_1\}$. Le STA $\mathcal{A}_{S_1}$ possède l'ensemble de transitions symboliques normalisées suivant :

$$\Delta = \{ \begin{array}{l} a \to X_0, b \to X_1, \\ s(X_0) \to X_1 \ \}. \end{array}$$ ◀

EXEMPLE *4.4*
Soit $\mathcal{A}_{S_2} = \langle \mathcal{F}, \mathcal{X}_\mathcal{Q}, \mathcal{X}^f_\mathcal{Q}, \Delta \rangle$ un STA, avec $\mathcal{F}$ un ensemble de symboles fonctionnels tel que $\mathcal{F} = \{p, r, nil_0, cons_2\}$, $\mathcal{X}_\mathcal{Q}$ et $\mathcal{X}^f_\mathcal{Q}$ deux ensembles d'états symboliques tels que $\mathcal{X}_\mathcal{Q} = \{X_p, X_r, X_{nil}, X_0, X_1\}$ et $\mathcal{X}^f_\mathcal{Q} = \{X_1\}$. Le STA $\mathcal{A}_{S_2}$ possède l'ensemble de transitions symboliques normalisées suivant :

$$\begin{aligned}\Delta = \{ \quad & p \to X_p, r \to X_r, nil \to X_{nil},\\ & cons(X_r, X_{nil}) \to X_0,\\ & cons(X_p, X_0) \to X_1 \ \}.\end{aligned}$$ ◀

Un STA possède donc des variables en guise d'états, qui peuvent être instanciées par des états. La définition suivante décrit comment un automate d'arbres classique peut être obtenu à partir d'un STA pour une instanciation donnée de l'ensemble $\mathcal{X}_\mathcal{Q}$ à un ensemble $\mathcal{Q}$ d'états.

DÉFINITION 4.3.3 (Instance d'un STA)
Soient $\mathcal{Q}$ un ensemble non-vide d'états, $\mathcal{A}_S = \langle \mathcal{F}, \mathcal{X}_\mathcal{Q}, \mathcal{X}^f_\mathcal{Q}, \Delta \rangle$ un STA, et $\mathcal{I}$ une instanciation telle que $\mathcal{I} : \mathcal{X}_\mathcal{Q} \mapsto \mathcal{Q}$. Une instance de $\mathcal{A}_S$ obtenue par l'instanciation $\mathcal{I}$, notée $\mathcal{A}^\mathcal{I}_S$, est un automate d'arbres classique $\langle \mathcal{F}, \mathcal{Q}^{\mathcal{A}^\mathcal{I}_S}, \mathcal{Q}^{\mathcal{A}^\mathcal{I}_S}_f, \Delta^{\mathcal{A}^\mathcal{I}_S} \rangle$, où :

- $\mathcal{Q}^{\mathcal{A}^\mathcal{I}_S} = \{\mathcal{I}(X) \mid X \in \mathcal{X}_\mathcal{Q}\}$; $\mathcal{Q}^{\mathcal{A}^\mathcal{I}_S}_f = \{\mathcal{I}(X) \mid X \in \mathcal{X}^f_\mathcal{Q}\}$;
- $\Delta^{\mathcal{A}^\mathcal{I}_S} = \{f(\mathcal{I}(X_1), \ldots, \mathcal{I}(X_n)) \to \mathcal{I}(X) \mid f(X_1, \ldots, X_n) \to X \in \Delta\}$.

EXEMPLE *4.5*
Soient $\mathcal{A}_{S_1}$ le STA défini dans l'exemple 4.3, $\mathcal{I}_1$ et $\mathcal{I}_2$ deux instanciations telles que $\mathcal{I}_1 = \{X_0 \mapsto q_0, X_1 \mapsto q_0\}$ et $\mathcal{I}_2 = \{X_0 \mapsto q_0, X_1 \mapsto q_1\}$. Alors $\mathcal{A}^{\mathcal{I}_1}_{S_1} = \langle \mathcal{F}, \{q_0\}, \{q_0\}, \Delta_1 \rangle$ et $\mathcal{A}^{\mathcal{I}_2}_{S_1} = \langle \mathcal{F}, \{q_0, q_1\}, \{q_1\}, \Delta_2 \rangle$, avec :

$$\begin{aligned}\Delta_1 = \{ \quad & a \to q_0, b \to q_0,\\ & s(q_0) \to q_0 \ \}, \text{ et}\\ \Delta_2 = \{ \quad & a \to q_0, b \to q_1,\\ & s(q_0) \to q_1 \ \}\end{aligned}$$

On peut alors remarquer que $\mathcal{L}(\mathcal{A}^{\mathcal{I}_1}_{S_1}) = \{s^n(a|b) \mid n \geq 0\}$ et $\mathcal{L}(\mathcal{A}^{\mathcal{I}_2}_{S_1}) = \{b, s(a)\}$. (voir Fig.4.3). ◀

EXEMPLE *4.6*
Soient $\mathcal{A}_{S_2}$ le STA défini dans l'exemple 4.4, $\mathcal{I}_1$ et $\mathcal{I}_2$ deux instanciations telles que $\mathcal{I}_1 = \{X_p \mapsto q_{pr}, X_r \mapsto q_{pr}, X_{nil} \mapsto q_{nil}, X_0 \mapsto q_0, X_1 \mapsto q_0\}$ et $\mathcal{I}_2 = \{X_p \mapsto q_p, X_r \mapsto q_r, X_{nil} \mapsto q_{nil}, X_0 \mapsto q_0, X_1 \mapsto q_1\}$. Alors $\mathcal{A}^{\mathcal{I}_1}_{S_2} = \langle \mathcal{F}, \{q_0\}, \{q_0\}, \Delta_1 \rangle$ et $\mathcal{A}^{\mathcal{I}_2}_{S_2} = \langle \mathcal{F}, \{q_1\}, \{q_1\}, \Delta_2 \rangle$, avec :

$$\begin{aligned}\Delta_1 = \{ \quad & p \to q_{pr}, r \to q_{pr},\\ & nil \to q_{nil}, cons(q_{pr}, q_{nil}) \to q_0,\\ & cons(q_{pr}, q_0) \to q_0 \ \}, \text{ et}\\ \Delta_2 = \{ \quad & p \to q_p, r \to q_r,\\ & nil \to q_{nil}, cons(q_r, q_{nil}) \to q_0,\\ & cons(q_p, q_0) \to q_1 \ \}\end{aligned}$$

On peut alors remarquer que $\mathcal{L}(\mathcal{A}^{\mathcal{I}_1}_{S_2}) = \{cons(p|r, nil), cons(p|r, cons(p|r, nil)), cons(p|r, cons(p|r, (cons(p|r, nil)))), \ldots\}$ et $\mathcal{L}(\mathcal{A}^{\mathcal{I}_2}_{S_2}) = \{cons(p, cons(r, nil))\}$. (voir Fig.4.4). ◀

Nous voulons maintenant définir la réduction d'un terme t dans un STA $\mathcal{A}_S$, soit la relation $t \to^*_{\mathcal{A}_S} X$, avec X un état symbolique de $\mathcal{A}_S$. Or les différentes réductions d'un terme peuvent être effectives ou non, selon les instanciations de $\mathcal{A}_S$. On veut alors pouvoir répondre à la question suivante : quelle instance du STA reconnait le terme t ? Les formules logiques de la définition 4.2.1 sont dans ce cas utilisées pour définir une propriété sur le langage. Ceci implique la définition d'un nouveau type de relation de

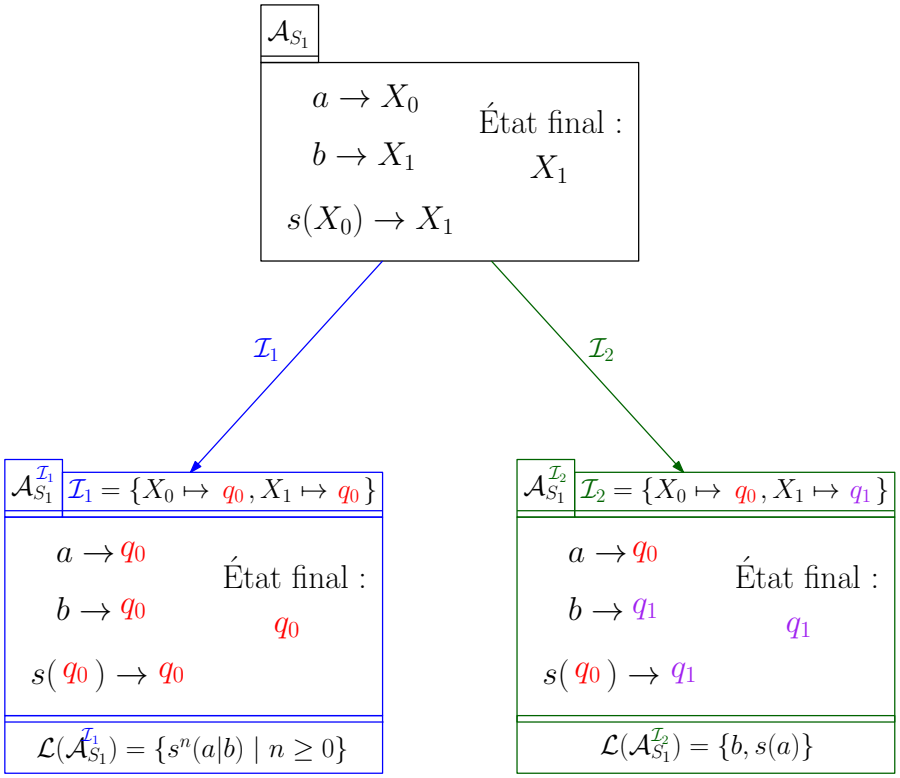


FIGURE 4.3 — Différentes instanciations d'un STA $\mathcal{A}_{S_1}$.

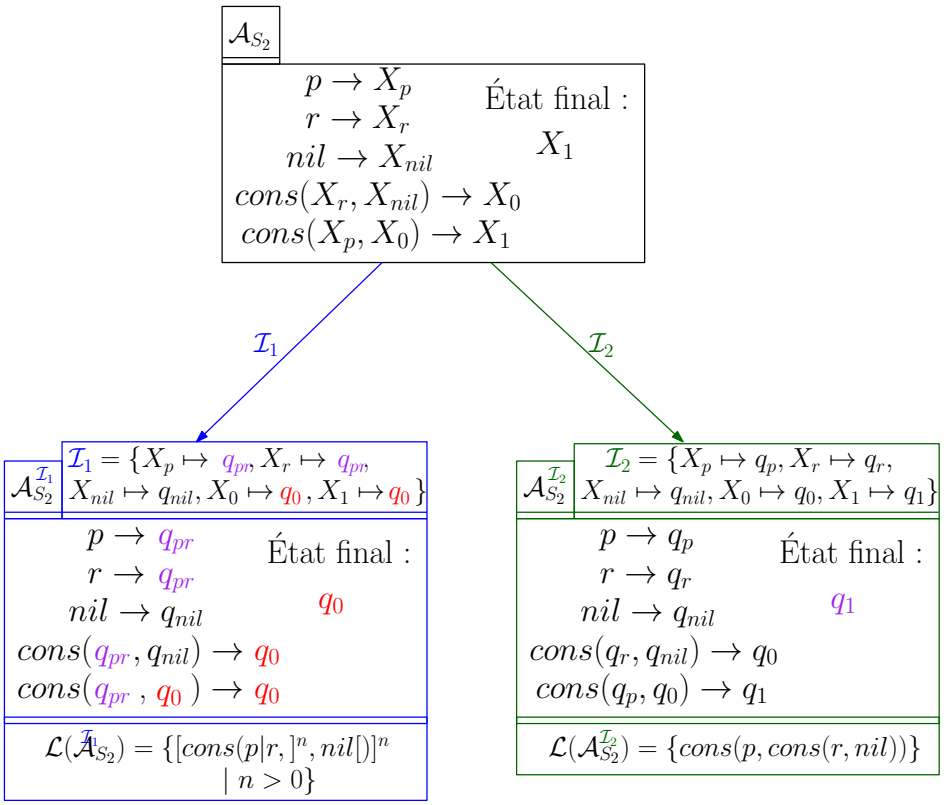


FIGURE 4.4 — Différentes instanciations d'un STA $\mathcal{A}_{S_2}$.

réduction, paramétrée par une formule logique de $\rho(\mathcal{X}_\mathcal{Q})$ conditionnant cette réduction.

La relation de réduction d'un STA est alors notée : $t \xrightarrow{\varphi}{}^*_{\mathcal{A}_S} X$, avec φ une formule de $\rho(\mathcal{X}_\mathcal{Q})$. Si une instanciation $\mathcal{I}$ de $\mathcal{A}_S$ satisfait φ alors la relation de réduction garantit que $\mathcal{A}_S^\mathcal{I}$ réduit le terme t sur l'état $\mathcal{I}(X)$. Remarquons que si $t \xrightarrow{\varphi}{}^*_{\mathcal{A}_S} X$ alors φ est une conjonction d'égalité entre états symboliques. Ceci est dû à une simple réduction du terme t utilisant les transitions de $\mathcal{A}_S$. Cette relation est la fermeture réflexive transitive de la relation de réécriture induite des transitions d'un STA, dont nous allons donner la définition.

DÉFINITION 4.3.4 ($t \xrightarrow{\varphi}_{\mathcal{A}_S} s$)
Soient $\mathcal{A}_S = \langle \mathcal{F}, \mathcal{X}_\mathcal{Q}, \mathcal{X}_\mathcal{Q}^f, \Delta \rangle$ un STA, t un terme de $\mathcal{T}(\mathcal{F}, \mathcal{X}_\mathcal{Q})$ et $X, Y, X_1, \ldots, X_n$ des états symboliques de $\mathcal{X}_\mathcal{Q}$. On a alors, pour toute position $p \in Pos(t)$:

- *Si $t|_p = X \in \mathcal{X}_\mathcal{Q}$, alors $X \xrightarrow{\top}_{\mathcal{A}_S} X$.*
- *Si $t|_p \in \mathcal{F}^0$ et $t|_p \to Y \in \Delta$, alors $t \xrightarrow{X=Y} t[X]_p$.*
- *Si $t|_p = f(X_1, \ldots, X_n)$ et $f(X_1, \ldots, X_n) \to Y \in \Delta$, alors $t \xrightarrow{X=Y} t[X]_p$.*
- *Si $t|_p = f(t_1, \ldots, t_n)$ et $t_1 \xrightarrow{\varphi_1} s_1, \ldots, t_n \xrightarrow{\varphi_n} s_n$, alors $t \xrightarrow{\varphi_1 \wedge \ldots \wedge \varphi_n} t[f(s_1, \ldots, s_n)]_p$.*

La relation de réduction $\xrightarrow{\varphi}{}^*_{\mathcal{A}_S}$ est alors la fermeture réflexive transitive de $\xrightarrow{\varphi}_{\mathcal{A}_S}$.

EXEMPLE *4.7*
Soit A_{S_1} le STA de l'exemple 4.3. Soit t un terme de $\mathcal{T}(\mathcal{F})$ tel que $t = s(s(a))$. Selon la définition 4.3.4, on a alors $t \xrightarrow{\varphi}{}^*_{\mathcal{A}_{S_1}} X_1$ avec $\varphi = X_0 = X_1$.

En effet $s(s(a)) \xrightarrow{\top}_{\mathcal{A}_{S_1}} s(s(X_0)) \xrightarrow{\top}_{\mathcal{A}_{S_1}} s(X_1) \xrightarrow{X_0=X_1}_{\mathcal{A}_{S_1}} X_1$, donc on a bien $s(s(a)) \xrightarrow{X_0=X_1}{}^*_{\mathcal{A}_{S_1}} X_1$. Soit $\mathcal{I}_1$ l'instanciation définie dans l'exemple 4.5. Selon la définition 4.2.2, on a $\mathcal{I}_1 \models \varphi$. On a alors $s(s(a)) \to^*_{\mathcal{A}_{S_1}^{\mathcal{I}_1}} \mathcal{I}_1(X_1) = q_0$. ◀

EXEMPLE *4.8*
Soit A_{S_2} le STA de l'exemple 4.4. Soit t un terme de $\mathcal{T}(\mathcal{F})$ tel que $t = cons(p, cons(r, cons(p, nil)))$. Selon la définition 4.3.4, on a alors $t \xrightarrow{\varphi}{}^*_{\mathcal{A}_{S_2}} X_1$ avec $\varphi = (X_p = X_r) \wedge (X_0 = X_1)$.

En effet on a $cons(p, cons(r, cons(p, nil))) \xrightarrow{\mathcal{X}_p=X_r}_{\mathcal{A}_{S_2}} cons(p, cons(r, cons(X_r, X_{nil}))) \xrightarrow{X_p=X_r}_{\mathcal{A}_{S_2}} cons(p, cons(X_r, X_0)) \xrightarrow{X_p=X_r}_{\mathcal{A}_{S_2}} cons(p, X_1) \xrightarrow{(X_p=X_r)\wedge(X_0=X_1)}_{\mathcal{A}_{S_2}} X_1$, donc on a bien $cons(p, cons(r, cons(p, nil))) \xrightarrow{(X_p=X_r)\wedge(X_0=X_1)}{}^*_{\mathcal{A}_{S_2}} X_1$. Soit $\mathcal{I}_1$ l'instanciation définie dans l'exemple 4.6. Selon la définition 4.2.2, on a $\mathcal{I}_1 \models \varphi$. On a alors $cons(p, cons(r, cons(p, nil))) \to^*_{\mathcal{A}_{S_2}^{\mathcal{I}_1}} \mathcal{I}_1(X_1) = q_0$. ◀

Pour un STA $\mathcal{A}_S$ donné, un terme t, un état symbolique X, et une instanciation $\mathcal{I}$, si *pour toute* formule φ telle que $t \xrightarrow{\varphi}{}^*_{\mathcal{A}_S} X$, on a $\mathcal{I} \not\models \varphi$, alors on peut conclure que $t \not\to^*_{\mathcal{A}_S^\mathcal{I}} \mathcal{I}(X)$.

La proposition suivante présente la caractérisation, par une formule de $\rho(\mathcal{X}_\mathcal{Q})$, de l'acceptation d'un terme t par un STA $\mathcal{A}_S$ donné. Par conséquent, pour chaque instanciation $\mathcal{I}$ de $\mathcal{A}_S$ telle que $\mathcal{I}$ satisfait cette formule, $\mathcal{A}_S^\mathcal{I}$ est un automate qui reconnait le terme t.

PROPOSITION 4.3.5 (Caractérisation de la reconnaissance d'un terme)
Soient $\mathcal{A}_S = \langle \mathcal{F}, \mathcal{X}_\mathcal{Q}, \mathcal{X}_\mathcal{Q}^f, \Delta \rangle$ un STA, $\mathcal{I}$ une instanciation, $t \in \mathcal{T}(\mathcal{F}, \mathcal{X}_\mathcal{Q})$ et $X \in \mathcal{X}_\mathcal{Q}$. Soit la formule suivante caractérisant la reconnaissance du terme t sur l'état X de $\mathcal{A}_S$:

$$\texttt{Reco}(t, X) = \bigvee_{t \xrightarrow{\varphi}^*_{\mathcal{A}_S} X} \varphi.$$

Alors on a :

$$\mathcal{I} \models \texttt{Reco}(t, X) \;\Leftrightarrow\; t\mathcal{I} \rightarrow^*_{\mathcal{A}_S^\mathcal{I}} \mathcal{I}(X).$$

Démonstration. Premièrement, on doit montrer que $\mathcal{I} \models \texttt{Reco}(t, X) \Rightarrow t\mathcal{I} \rightarrow^*_{\mathcal{A}_S^\mathcal{I}} \mathcal{I}(X)$. On peut remarquer par construction que la formule $\texttt{Reco}(t, X)$ est une forme normale disjonctive. En effet, une formule φ impliquée dans une une réduction telle que $t \xrightarrow{\varphi}^*_{\mathcal{A}_S} X$ est une conjonction d'atomes $X = Y$ ou $X \neq Y$ selon la définition 4.3.4. Comme $\mathcal{I} \models \texttt{Reco}(t, X)$, alors selon la définition 4.2.2, on peut déduire qu'il existe une sous formule φ_i telle que $\texttt{Reco}(t, X) = \varphi_1 \vee \ldots \vee \varphi_i \vee \ldots \vee \varphi_n$, $t \xrightarrow{\varphi_i}^*_{\mathcal{A}_S} X$ et $\mathcal{I} \models \varphi_i$. Procédons par induction sur le terme t.

Cas de base :

- Si t est une constante : étant donné que $t \xrightarrow{\varphi_i}^*_{\mathcal{A}_S} X$, alors il existe une transition de la forme $t \rightarrow X' \in \Delta$ selon la définition 4.3.4. Par conséquent, $\varphi_i = (X = X')$. Comme $\mathcal{I} \mapsto \varphi_i$ et $\varphi_i = (X = X')$, $\mathcal{I}(X) = \mathcal{I}(X')$, on a $t\mathcal{I} \rightarrow_{\mathcal{A}_S^\mathcal{I}} \mathcal{I}(X)$.
- Si t est une état symbolique de $\mathcal{X}_\mathcal{Q}$: étant donné que nous avons $t \xrightarrow{\varphi_i}^*_{\mathcal{A}_S} X$, alors $t = X$ et $t \xrightarrow{\top}^*_{\mathcal{A}_S} X$ selon la définition 4.3.4. Par conséquent, $\varphi_i = \top$, et comme $\mathcal{I} \models \varphi_i$, on peut déduire que $t\mathcal{I} \rightarrow_{\mathcal{A}_S^\mathcal{I}} \mathcal{I}(X)$.

Induction:
Supposons maintenant que t soit un terme de la forme $f(t_1, \ldots, t_n)$. Par hypothèse, $t \xrightarrow{\varphi_i}^*_{\mathcal{A}_S} X$. On peut déduire de la définition 4.3.4 qu'il existe $f(Y_1, \ldots, Y_n) \rightarrow X' \in \Delta$ et $\psi_1, \ldots \psi_n$ tels que:

1. pour chaque t_j, on a $t_j \xrightarrow{\psi_j} Y_j$ et
2. $\varphi_i = \psi_1 \wedge \ldots \wedge \psi_n$.

Par hypothèse d'induction, pour chaque t_j, Y_j et $\texttt{Reco}(t_j, Y_j)$, on a :

$$\mathcal{I} \models \texttt{Reco}(t_j, Y_j) \Leftrightarrow t_j\mathcal{I} \rightarrow^*_{\mathcal{A}_S^\mathcal{I}} \mathcal{I}(Y_j), \tag{4.1}$$

avec $\texttt{Reco}(t_j, Y_j) = \bot \vee \bigvee_{t_j \xrightarrow{\phi} Y_j} \phi$. Comme $t_j \xrightarrow{\psi_i} Y_j$ et $\mathcal{I} \models \psi_j$, on peut déduire de façon triviale que $\mathcal{I} \models \texttt{Reco}(t_j, Y_j)$. Donc, en appliquant (4.1), on a alors, pour tout t_j, $t_j\mathcal{I} \rightarrow^*_{\mathcal{A}_S^\mathcal{I}} \mathcal{I}(Y_j)$. Selon la définition 4.3.3, on peut déduire que :

$$f(\mathcal{I}(Y_1), \ldots, \mathcal{I}(Y_n)) \rightarrow \mathcal{I}(X') \text{ est une transition de } \mathcal{A}_S^\mathcal{I}. \tag{4.2}$$

Par conséquent, on peut déduire que $t\mathcal{I} = f(t_1\mathcal{I}, \ldots, t_1\mathcal{I}) \rightarrow^*_{\mathcal{A}_S^\mathcal{I}} f(\mathcal{I}(Y_1), \ldots, \mathcal{I}(Y_n))$ à partir de $t_j\mathcal{I} \rightarrow^*_{\mathcal{A}_S^\mathcal{I}} \mathcal{I}(Y_j)$. De plus, en utilisant la transition (4.2), on a $t\mathcal{I} \rightarrow^*_{\mathcal{A}_S^\mathcal{I}} \mathcal{I}(X'_i)$. Comme $\mathcal{I}(X'_i) = \mathcal{I}(X)$, on peut déduire que $t\mathcal{I} \rightarrow^*_{\mathcal{A}_S^\mathcal{I}} \mathcal{I}(X)$.

Nous devons alors montrer que $\mathcal{I} \models \texttt{Reco}(t, X) \Leftarrow t\mathcal{I} \rightarrow^*_{\mathcal{A}_S^{\mathcal{I}}} \mathcal{I}(X)$. Procédons par récurrence sur la structure de t.

Cas de base :

- Si t est une constante : par hypothèse, $t\mathcal{I} \rightarrow^*_{\mathcal{A}_S^{\mathcal{I}}} \mathcal{I}(X)$. Alors dans ce cas, on a $t\mathcal{I} = t$. D'après la définition 4.3.3, il existe une transition $t \rightarrow X' \in \Delta$ telle que $\mathcal{I}(X) = \mathcal{I}(X')$. Par conséquent, d'après la définition 4.2.2, on peut déduire que $t \xrightarrow{X=X'}{}^*_{\mathcal{A}_S} X$ et que $\mathcal{I} \models X = X'$. Alors, comme $\texttt{Reco}(t, X) = \bigvee_{t \xrightarrow{\varphi}{}^*_{\mathcal{A}_S} X} \varphi = \varphi_1 \vee \ldots \vee (X = X') \vee \ldots \varphi_n$, on a donc $\mathcal{I} \models \texttt{Reco}(t, X)$.
- Si $t\mathcal{I}$ est un état : on a alors nécessairement $t\mathcal{I} = \mathcal{I}(X)$. Selon la définition 4.3.4, on a $X' \xrightarrow{\varphi}{}^*_{\mathcal{A}_S} X$ si et seulement si $X' = X$ et $\varphi = \top$. Par conséquent, $\texttt{Reco}(X, X) = \top$, donc on peut déduire que $\mathcal{I} \models \texttt{Reco}(t, X)$ (d'après la définition 4.2.2).

Induction:
Supposons que t soit un terme de la forme $f(t_1, \ldots, t_n)$: Par hypothèse, $f(t_1, \ldots, t_n)\mathcal{I} \rightarrow^*_{\mathcal{A}_S^{\mathcal{I}}} \mathcal{I}(X)$. Alors il existe $X_1, \ldots, X_n, X' \in \mathcal{X}_{\mathcal{Q}}$ tels que

$$t_i\mathcal{I} \rightarrow^*_{\mathcal{A}_S^{\mathcal{I}}} \mathcal{I}(X_i),\ \mathcal{I}(X') = \mathcal{I}(X)\ et\ f(X_1, \ldots, X_n) \rightarrow X' \in \Delta. \tag{4.3}$$

Pour chaque t_i, on peut appliquer l'hypothèse d'induction. On obtient alors que $\mathcal{I} \models \texttt{Reco}(t_k, X_k)$, pour $k = 1, \ldots, n$. Regardons l'hypothèse d'induction : pour chaque t_k: $\mathcal{I} \models \texttt{Reco}(t_k, X_k)$ si et seulement si $t_k\mathcal{I} \rightarrow^*_{\mathcal{A}_S^{\mathcal{I}}} \mathcal{I}(X_k)$ avec $\texttt{Reco}(t_k, X_k) = \vee \bigvee_{\{t \xrightarrow{\psi}{}_{\mathcal{A}_S} X_k\}} \psi$. Comme $\mathcal{I} \models \texttt{Reco}(t_k, X_k)$, il existe alors une sous-formule de $\texttt{Reco}(t_k, X_k)$, *i.e.* ψ_k, telle que $\mathcal{I} \models \psi_k$. Par conséquent, $f(t_1, \ldots, t_n) \xrightarrow{\psi_1}{}_{\mathcal{A}_S} f(X_1, t_2, \ldots, t_n)$. En itérant le processus, on obtient alors que $f(t_1, \ldots, t_n) \xrightarrow{\psi_1 \wedge \ldots \wedge \psi_n}{}_{\mathcal{A}_S} f(X_1, \ldots, X_n)$. Comme $f(X_1, \ldots, X_n) \rightarrow X' \in \Delta$, on peut déduire que $t \xrightarrow{\varphi}{}_{\mathcal{A}_S} X$ avec $\varphi = \psi_1 \wedge \ldots \wedge \psi_n \wedge X' = X$. D'après la déduction (4.3) et la définition 4.2.2, on a alors $\mathcal{I} \models \varphi$. Ce qui nous permet de conclure cette preuve.

□

Notons que la formule $\texttt{Reco}(t, X)$ est forcément finie. En effet, elle est une disjonction de toutes les formules ϕ possibles conditionnant la réduction du terme t sur l'état symbolique X (*i.e.* telles que $t \xrightarrow{\varphi}{}^*_{\mathcal{A}_S} X$). Or le nombre de ces réductions est fini. En effet, d'après la définition 4.3.4 de la réduction, le nombre de réductions possibles dépend des transitions de l'automate, qui sont en nombre fini. Une réduction dans le cas des automates d'arbres classique est toujours finie (voir [Genet, 2009]). Le cas des STA construit des formules logiques lors de la réduction et augmente le nombre de réductions possibles, mais ce nombre est fini. De plus, une réduction s'opère d'une manière similaire au cas classique, et termine donc toujours.

4.4 Solutions de l'algorithme de filtrage pour les STA

Rappelons que nous voulons ici caractériser, par une formule logique, une approximation point-fixe qui n'admet pas de terme interdit, *i.e.* qui vérifie une propriété de

sûreté. Un automate d'arbres est un point-fixe par rapport à un système de réécriture $\mathcal{R}$ si ce dernier ne permet plus de rajouter de nouvelles transitions à l'automate. On dit alors que l'automate est $\mathcal{R}$-clos (voir définition 2.4.1). Soit $\mathcal{X}$ l'ensemble de variables d'un système de réécriture et $\mathcal{X}_\mathcal{Q}$ l'ensemble d'états symboliques du STA. Il est à noter que dans la suite de ce chapitre, $\mathcal{X}$ sera toujours disjoint de $\mathcal{X}_\mathcal{Q}$.

Un automate $\mathcal{A} = \langle \mathcal{F}, \mathcal{Q}, \mathcal{Q}^f, \Delta \rangle$ est $\mathcal{R}$-clos si l'algorithme de complétion n'ajoute plus aucune transition à l'automate. Pour chaque règle $l \to r$ du système de réécriture $\mathcal{R}$, pour chaque substitution $\sigma : \mathcal{X} \mapsto \mathcal{Q}$, si $l\sigma \to^*_\mathcal{A} q$ alors on ajoute les transitions nécessaires pour avoir $r\sigma \to^* q$ dans Δ, si cette dernière réduction n'existe pas déjà dans l'automate (voir section 2.4.1). Pour pouvoir savoir si un automate $\mathcal{A}$ est $\mathcal{R}$-clos, il faut donc faire le calcul de toutes les substitutions σ possibles telles que $l\sigma \to^*_\mathcal{A} q$. Dans cette section, nous proposons donc dans un premier temps une adaptation de l'algorithme de filtrage (calcul des substitutions) au cas des STA.

Soit t un terme de $\mathcal{T}(\mathcal{F}, \mathcal{X})$. Pour un automate d'arbres classique $\mathcal{A}$ et un état q, l'algorithme de filtrage $t \trianglelefteq q$ possède une solution s'il existe une substitution $\sigma : \mathcal{X} \mapsto \mathcal{Q}$ telle que $t\sigma \to^*_\mathcal{A} q$. Dans le cas des STA, l'algorithme de filtrage est formalisé sur les états symboliques plutôt que sur les états classiques, *i.e.* $t \trianglelefteq X$ avec $X \in \mathcal{X}_\mathcal{Q}$. En effet, dans ce contexte et si l'on considère un STA $\mathcal{A}_S$, les solutions de l'algorithme sont représentées comme des ensembles de paires (σ, φ) avec σ une substitution de $\mathcal{X}$ à $\mathcal{X}_\mathcal{Q}$ et φ une formule de $\rho(\mathcal{X}_\mathcal{Q})$ telle que $t\sigma \xrightarrow{\varphi}^*_{\mathcal{A}_S} X$. Supposons $\mathcal{I} : \mathcal{X}_\mathcal{Q} \mapsto \mathcal{Q}$ une instanciation. D'un point de vue sémantique, une solution (φ, σ) signifie que, si l'on a $\mathcal{I} \models \varphi$, alors la substitution $\sigma \circ \mathcal{I}$ est une solution du problème de filtrage $t \trianglelefteq \mathcal{I}(X)$ dans l'automate d'arbres $\mathcal{A}^\mathcal{I}_S$.

DÉFINITION 4.4.1 (Algorithme de filtrage (ou *matching*) – S^t_X)
Soit $\mathcal{A}_S = \langle \mathcal{F}, \mathcal{X}_\mathcal{Q}, \mathcal{X}^f_\mathcal{Q}, \Delta \rangle$ un STA. On note S^t_X l'ensemble des solutions possibles au problème de filtrage $t \trianglelefteq X$, avec t un terme linéaire et X un état symbolique de $\mathcal{X}_\mathcal{Q}$. S^t_X est calculé grâce à l'application successive des règles de déductions de la Fig.4.5, en donnant le couple $(t \trianglelefteq X, \top)$ comme entrée de l'algorithme.

EXEMPLE *4.9*
Soit $\mathcal{A}_{S_1}$ le STA de l'exemple 4.3. Nous rappelons que $\mathcal{A}_{S_1}$ possède l'ensemble de transitions suivant :

$$\begin{aligned}\Delta = \{ \quad & a \to X_0, b \to X_1, \\ & s(X_0) \to X_1 \ \}.\end{aligned}$$

En utilisant les règles de déduction, on a par exemple : $S^a_{X_0} = \{(\emptyset, X_0 = X_0)\}$. En effet, $a \to X_0 \in \Delta$ et il suffit d'appliquer la règle $(Constante)$. Ensuite, comme exposé sur la figure 4.6, on a également : $S^{s(s(x))}_{X_1} = \{(x \mapsto X_0, X_0 = X_1)\}$. ◀

EXEMPLE *4.10*
Soit $\mathcal{A}_{S_2}$ le STA de l'exemple 4.4. Nous rappelons que $\mathcal{A}_{S_2}$ possède l'ensemble de transitions suivant :

$$\begin{aligned}\Delta = \{ \quad & p \to X_p, r \to X_r, nil \to X_{nil}, \\ & cons(X_r, X_{nil}) \to X_0, \\ & cons(X_p, X_0) \to X_1 \ \}.\end{aligned}$$

(Variable)

$$\frac{(t = x \trianglelefteq X, \varphi)}{\{(\{x \mapsto X\}, \varphi)\}} \qquad (x \in \mathcal{X})$$

(Constante)

$$\frac{(t = a \trianglelefteq X, \varphi)}{\bigoplus_{\forall a \to Y \in \Delta}(\emptyset, \varphi \wedge (X = Y))} \qquad (a \in \mathcal{F}^0)$$

(État Symbolique)

$$\frac{(t = Y \trianglelefteq X, \varphi)}{(\emptyset, \varphi \wedge (X = Y))} \qquad (Y \in \mathcal{X}_{\mathcal{Q}})$$

(Configuration)

$$\frac{(t = f(t_1, \ldots, t_n) \trianglelefteq X, \varphi)}{\bigoplus_{\forall f(X_1,\ldots,X_n) \to Y \in \Delta}(f(t_1, \ldots, t_n) \trianglelefteq f(X_1, \ldots, X_n), \varphi \wedge (X = Y)) \oplus \bot} \qquad (f \in \mathcal{F}^n)$$

(Décomposition)

$$\frac{(t = f(t_1, \ldots, t_n) \trianglelefteq f(X_1, \ldots, X_n), \varphi)}{(t_1 \trianglelefteq X_1, \varphi) \otimes \ldots \otimes (t_n \trianglelefteq X_n, \varphi)} \qquad (f \in \mathcal{F}^n)$$

(Normalisation)

$$\frac{(\sigma_1, \varphi_1) \otimes (\sigma_2, \varphi_2)}{(\sigma_1 \cup \sigma_2, \varphi_1 \wedge \varphi_2)} \quad , \quad \frac{(\sigma_1, \varphi_1) \oplus (\sigma_2, \varphi_2)}{\{(\sigma_1, \varphi_1), (\sigma_2, \varphi_2)\}} \quad ,$$

$$\frac{(\sigma, \varphi) \oplus \bot}{(\sigma, \varphi)} \quad , \quad \frac{(\sigma, \varphi) \otimes \bot}{\bot} \quad , et$$

$$\frac{((\sigma_1, \varphi_1) \oplus (\sigma_2, \varphi_2)) \otimes ((\sigma_3, \varphi_3) \oplus (\sigma_4, \varphi_4))}{((\sigma_1, \varphi_1) \otimes (\sigma_3, \varphi_3)) \oplus ((\sigma_1, \varphi_1) \otimes (\sigma_4, \varphi_4)) \oplus ((\sigma_2, \varphi_2) \otimes (\sigma_3, \varphi_3)) \oplus ((\sigma_2, \varphi_2) \otimes (\sigma_4, \varphi_4))}$$

FIGURE 4.5 — Règles de déductions de l'algorithme de filtrage : calcul de S_X^t.

$$
\begin{array}{rcl}
\text{(Configuration)} & \dfrac{(s(s(x)) \trianglelefteq X_1, X_0 = X_0)}{(s(s(x)) \trianglelefteq s(X_0), X_0 = X_0)} & (s(X_0) \to X_1 \in \Delta) \\
\text{(Décomposition)} & \dfrac{}{(s(x) \trianglelefteq X_0, X_0 = X_0)} & \\
\text{(Configuration)} & \dfrac{}{(s(x) \trianglelefteq s(X_0), (X_0 = X_0) \wedge (X_0 = X_1))} & (s(X_0) \to X_1 \in \Delta) \\
\text{(Décomposition)} & \dfrac{}{(x \trianglelefteq X_0, (X_0 = X_0) \wedge (X_0 = X_1))} & \\
\text{(Variable)} & \dfrac{}{(\{x \mapsto X_0\}, (X_0 = X_0) \wedge (X_0 = X_1))} & (x \in \mathcal{X})
\end{array}
$$

FIGURE 4.6 — Application de l'algorithme de filtrage pour le calcul de $S_{X_1}^{s(s(x))}$ (Ex.4.9).

En utilisant les règles de déduction comme exposé sur la figure 4.7, on a alors : $S_{X_0}^{cons(r,cons(x,nil))} = \{(x \mapsto X_r, (X_0 = X_{nil}) \vee ((X_0 = X_1) \wedge (X_p = X_r)), (x \mapsto X_p, ((X_0 = X_{nil}) \wedge (X_1 = X_{nil})) \vee ((X_0 = X_{nil}) \wedge (X_0 = X_1) \wedge (X_p = X_r))\}$.

◀

PROPOSITION 4.4.2 (Terminaison de l'algorithme de filtrage)
Soient $\mathcal{A}_S$ un STA, X un état symbolique de $\mathcal{X}_Q$, t un terme linéaire de $\mathcal{T}(\mathcal{F}, \mathcal{X})$ Alors l'ensemble des solutions possibles au problème de filtrage $t \trianglelefteq X$ dans $\mathcal{A}_S$, noté S_X^t, est fini, et son calcul via l'algorithme de filtrage est terminant.

Démonstration. Il s'agit de prouver que l'algorithme de filtrage termine sur tout problème initial $(t \trianglelefteq X, \top)$. La preuve va s'organiser comme suit : tout d'abord, il faut prouver que chaque règle, i.e. chaque déduction, est terminante (1). Ensuite, nous allons prouver que le calcul de la formule logique φ est également terminant (2). Puis, afin de prouver la terminaison de l'algorithme de filtrage, il faut alors prouver que le littéral gauche t du problème $(t \trianglelefteq X, \top)$ se réduit petit à petit (3).

(1) Chaque règle de déduction de la figure 4.5, prise séparément, termine simplement. On peut le voir de façon évidente sur les règles (Variable), (État Symbolique), (Décomposition) et (Normalisation). Les règles (Constantes) et (Configuration) terminent également simplement. En effet, ces règles créent une disjonction $\oplus$ de tuples $(t \trianglelefteq t', \varphi)$, et le nombre de tuples créé est nécessairement fini puisqu'il dépend d'un nombre de transitions ($\forall a \to Y \in \Delta$ ou $\forall f(X_1, \ldots, X_n) \to Y \in \Delta$), et que le nombre de transitions de Δ est fini.
(2) Pour chaque substitution conditionnée (σ, φ) calculée grâce à l'algorithme, la taille de la formule logique φ générée est également finie. En effet, les règles (Variable) et (Décomposition) conservent la taille de φ. Les règles (État symbolique), (Constantes) et (Configuration) lui ajoutent une seule conjonction "$\wedge(X = Y)$", dépendant d'une transition telle que $a \to Y \in \Delta$ ou $f(X_1, \ldots, X_n) \to Y \in \Delta$. La première règle de (Normalisation) créée des conjonctions de plusieurs formules (*i.e.* le tuple $(\sigma_1 \cup \sigma_2, \varphi_1 \wedge \varphi_2)$ dans le cas où on a une conjonction $\otimes$: $(\sigma_1, \varphi_1) \otimes (\sigma_2, \varphi_2)$). Or la conjonction $\otimes$ est créée lors de la règle (Décomposition), et cette règle génère un nombre fini de conjonction $\otimes$. Donc la formule φ est finie et le calcul de la formule φ est terminant.

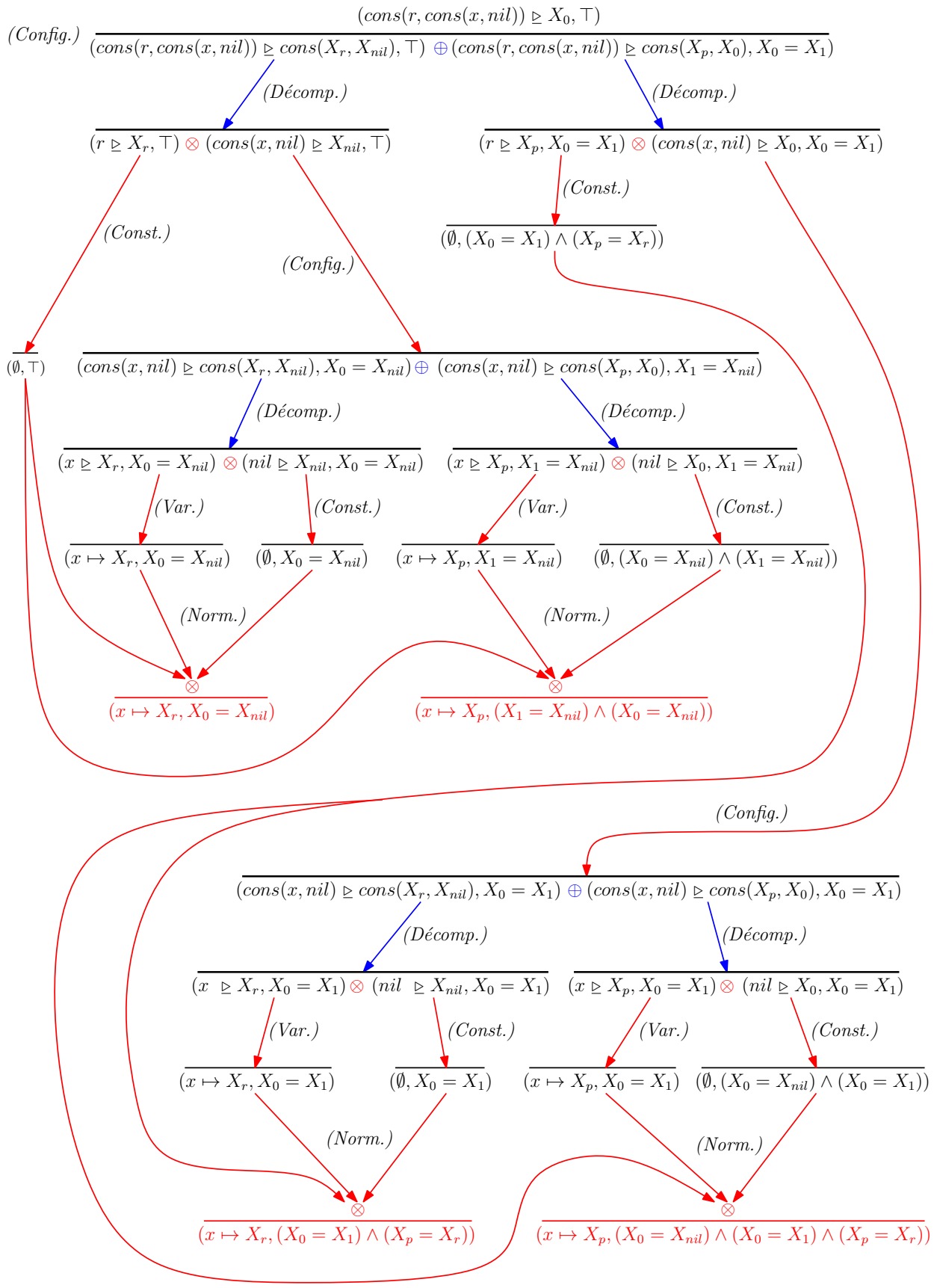


FIGURE 4.7 — Application de l'algorithme de filtrage pour le calcul de $S_{X_0}^{cons(r,cons(x,nil))}$ (exemple 4.10).

(3) Étant donné que nous avons prouvé que chaque règles, prise séparément, terminent simplement, ainsi que le calcul de φ, il faut ensuite prouver que l'algorithme de filtrage est terminant.Considérons la taille des membres gauches des littéraux $t \trianglelefteq X$ d'un problème de filtrage (*i.e.* t). Pour que l'algorithme de filtrage termine, il faut que la taille de t aille en décroissant jusqu'à ne plus pouvoir être réduite par les règles de déduction. Les règles (Variable) et (État symbolique) transforment les littéraux réduits en substitutions (σ, φ). Les règles de (Normalisation) n'agissent que sur les littéraux transformés en substitutions (σ, φ') donc également après réduction des littéraux $(t \trianglelefteq t', \varphi)$. Les règles (Décomposition) et (Constante) font décroitre strictement la taille de t et de t', alors que la règle (Configuration) la conserve (sauf dans le cas où il n'y a aucune transitions de Δ telle que $f(X_1, \ldots, X_n) \to Y$). Cependant, si l'on applique (Configuration) sur un problème de filtrage $(t \trianglelefteq X, \top)$ donné, on obtient respectivement une disjonction $\oplus$ finie de littéraux $(t_1 \trianglelefteq t'_1, \varphi_1) \oplus \ldots \oplus (t_p \trianglelefteq t'_p, \varphi_p) \oplus \bot$. Sur chaque littéral $(t_i \trianglelefteq t'_i, \varphi_i)$, la seule règle pouvant être appliquée est (Décomposition). En d'autres termes, chaque pas d'application de la règle (Configuration) est nécessairement suivi d'au moins une application de la règle (Décomposition), et fait donc strictement décroitre la taille des membres gauches. L'algorithme de filtrage est donc terminant.

□

La proposition suivante démontre que cet algorithme est correct, complet et terminant.

PROPOSITION 4.4.3 (Correction, complétude et terminaison)
Soient $\mathcal{A}_S$ un STA $\langle \mathcal{F}, \mathcal{X}_\mathcal{Q}, \mathcal{X}^f_\mathcal{Q}, \Delta \rangle$, X un état symbolique de $\mathcal{X}_\mathcal{Q}$, t un terme linéaire de $\mathcal{T}(\mathcal{F}, \mathcal{X})$ et $\sigma : \mathcal{V}ar(t) \mapsto \mathcal{X}_\mathcal{Q}$ une substitution. On a alors :

$$t\sigma \xrightarrow{\varphi}{}^*_{\mathcal{A}_S} X \text{ si et seulement si } (\sigma, \varphi) \in S^t_X,$$

*où S^t_X est un ensemble **fini** représentant l'ensemble des solutions du problème de filtrage $t \trianglelefteq X$.*

Démonstration. Il s'agit de prouver l'équivalence suivante : $\forall(\sigma, \varphi),\ t\sigma \xrightarrow{\varphi}{}^*_{\mathcal{A}_S} X$ si et seulement si $(\sigma, \varphi) \in S^t_X$. Soient $t \in \mathcal{T}(\mathcal{F}, \mathcal{X})$, $X \in \mathcal{X}_\mathcal{Q}$, $\varphi \in \rho(\mathcal{X}_\mathcal{Q})$ et $\sigma : \mathcal{X} \mapsto \mathcal{X}_\mathcal{Q}$ et $\mathcal{A}_S$ un STA, tels que $t\sigma \xrightarrow{\varphi}_{\mathcal{A}_S} X$, prouvons d'abord que $t\sigma \xrightarrow{\varphi}{}^*_{\mathcal{A}_S} X \implies (\sigma, \varphi) \in S^t_X$. Procédons par récurrence.

Cas de base :

- Si $t\sigma$ est une constante : par hypothèse, $t\sigma \xrightarrow{\varphi}{}^*_{\mathcal{A}_S} X$. Comme t est une constante, on a alors $\mathcal{V}ar(t) = \emptyset$, ainsi que $\sigma = \emptyset$. Il existe donc une transition $t \to Y \in \Delta$ et $\varphi = (X = Y)$. En appliquant la règle de déduction *(Constante)* avec la transition $t \to Y$ et le couple d'entrée $(t \trianglelefteq X, \top)$, on a alors $(\emptyset, X = Y) \in S^t_X$.

- Si $t\sigma$ est un état symbolique : selon la définition 4.3.4, $t\sigma \xrightarrow{\varphi}{}^*_{\mathcal{A}_S} X$ seulement si $t\sigma = X$ et $\varphi = \top$. De plus, si $t\sigma \in \mathcal{X}_\mathcal{Q}$ alors $t \in \mathcal{X}$. Par conséquent, on peut définir la substitution $\sigma : \{t \mapsto X\}$. Calculons l'ensemble des solutions S^t_X. Rappelons que $t \in \mathcal{X}$. Alors, selon la règle (Variable) de l'algorithme de filtrage appliquée sur le couple d'entrée $(t \trianglelefteq X, \top)$, on obtient alors que $((t \mapsto X), \top) \in S^t_X$.

Induction :

Supposons t un terme de la forme $f(t_1, \ldots, t_n)$. Étant donné que t est linéaire, on peut alors décomposer σ en n substitutions $\sigma_1 : \mathcal{V}ar(t_1) \mapsto \mathcal{X}_\mathcal{Q}$, ..., $\sigma_n : \mathcal{V}ar(t_n) \mapsto \mathcal{X}_\mathcal{Q}$. Alors, $\sigma(x) = \sigma_i(x)$ si $x \in \mathcal{V}ar(t_i)$. Autrement dit, $\sigma = \sigma_1 \cup \ldots \cup \sigma_n$ (1). On a donc $t\sigma = f(t_1\sigma_1, \ldots, t_n\sigma_n)$. Par hypothèse, on a $t\sigma \xrightarrow{\varphi}{}^*_{\mathcal{A}_S} X$. En utilisant la définition 4.3.4, on peut alors déduire qu'il existe $\varphi_1, \ldots, \varphi_n \in \rho(\mathcal{X}_\mathcal{Q})$ tels que $t\sigma \xrightarrow{\varphi'}{}^*_{\mathcal{A}_S} f(X_1, \ldots, X_n)$ et $t_1 \xrightarrow{\varphi_1}{}^*_{\mathcal{A}_S} X_1, \ldots, t_n \xrightarrow{\varphi_n}{}^*_{\mathcal{A}_S} X_n$, avec $\varphi' = \varphi_1 \wedge \ldots \wedge \varphi_n$ (a). Toujours en utilisant cette définition, on peut ensuite déduire qu'il existe une transition $f(X_1, \ldots, X_n) \to Y \in \Delta$ telle que $f(X_1, \ldots, X_n) \xrightarrow{X=Y}{}^*_{\mathcal{A}_S} X$ (b). De (a) et de (b) on peut alors déduire que $t\sigma \xrightarrow{\varphi}{}^*_{\mathcal{A}_S} X$ si et seulement si $\varphi = \varphi_1 \wedge \ldots \wedge \varphi_n \wedge X = Y$ (2) et $t_i\sigma_i \xrightarrow{\varphi_i}{}^*_{\mathcal{A}_S} X_i$, pour tout $i \in [1, n]$.

En appliquant la règle (Configuration) avec la transition $f(X_1, \ldots, X_n) \to Y$ et le couple d'entrée $(f(t_1, \ldots, t_n) \trianglelefteq X, \top)$, on obtient alors $(f(t_1, \ldots, t_n) \trianglelefteq f(X_1, \ldots, X_n), X = Y)$. On peut alors ensuite appliquer la règle (Décomposition) et on a alors $(t_1 \trianglelefteq X_1, X = Y) \otimes \ldots \otimes (t_n \trianglelefteq X_n, X = Y)$ (3). D'après l'hypothèse, $t_i\sigma_i \xrightarrow{\varphi_i}{}^*_{\mathcal{A}_S} Y_i$ pour tout $i \in [1, n]$, donc d'après l'hypothèse d'induction, on a $(\sigma_i, \varphi_i) \in S^{t_i}_{X_i}$. L'algorithme va donc poursuivre depuis (3) jusqu'à obtenir $(\sigma_1 \trianglelefteq X_1, \varphi_1 \wedge (X = Y)) \otimes \ldots \otimes (\sigma_n \trianglelefteq X_n, \varphi_n \wedge (X = Y))$. En appliquant (Normalisation), on obtient alors $(\sigma_1 \cup \ldots \cup \sigma_n, \varphi_1 \wedge \ldots \wedge \varphi_n \wedge (X = Y))$.

Or, d'après (1), $\sigma_1 \cup \ldots \cup \sigma_n = \sigma$, et d'après (2), $\varphi_1 \wedge \ldots \wedge \varphi_n \wedge (X = Y) = \varphi$. On peut donc déduire que $(\sigma, \varphi) \in S^t_X$.

Soient $t \in \mathcal{T}(\mathcal{F}, \mathcal{X})$, $X \in \mathcal{X}_\mathcal{Q}$, $\varphi \in \rho(\mathcal{X}_\mathcal{Q})$ et $\sigma : \mathcal{X} \mapsto \mathcal{X}_\mathcal{Q}$ et $\mathcal{A}_S$ un STA, tels que $(\sigma, \varphi) \in S^t_X$, prouvons maintenant l'autre sens, *i.e.* que $(\sigma, \varphi) \in S^t_X \implies t\sigma \xrightarrow{\varphi}{}^*_{\mathcal{A}_S} X$. Procédons par récurrence.

Cas de base :

- Si t est une constante, le calcul de S^t_X démarre avec le problème de filtrage $(t \trianglelefteq X, \top)$ et applique alors la règle (Constante). On a une conjonction $\oplus$ de substitutions (σ_i, φ_i) calculée : pour toute transition $t \to Y \in \Delta$, cette règle génère une substitution $(\emptyset, \top \wedge (X = Y))$. Comme $t \to Y \in \Delta$, on a bien $t\sigma = t \xrightarrow{X=Y}{}^*_{\mathcal{A}_S} X$ d'après la définition 4.3.4.
- Si t est une variable de $\mathcal{X}$, on applique la règle (Variable) sur le problème de filtrage $(t \trianglelefteq X, \top)$, et on obtient la substitution $(\{t \mapsto X\}, \top)$. Or on a vu dans la définition 4.3.4 que $X \xrightarrow{\top}_{\mathcal{A}_S} X$, donc $t\sigma = X \xrightarrow{\top}{}^*_{\mathcal{A}_S} X$.

Induction : Supposons t un terme de la forme $f(t_1, \ldots, t_n)$. Pour calculer S^t_X depuis le problème de filtrage $(t = f(t_1, \ldots, t_n) \trianglelefteq X, \top)$, on applique d'abord la règle (Décomposition), et on obtient, pour toute transition $f(X_1, \ldots, X_n) \to Y \in \Delta$ (1), le couple $(f(t_1, \ldots, t_n), \top \wedge (X = Y))$. Depuis ce couple, il faut ensuite appliquer la règle (Décomposition) qui renvoie la conjonction $\otimes$ suivante : $(t_1 \trianglelefteq X_1, \top \wedge (X = Y)) \otimes \ldots \otimes (t_n \trianglelefteq X_n, \top(X = Y))$ (2). Chaque couple $(t_i \trianglelefteq X_i, \top)$ passé en entrée de l'algorithme de filtrage permet le calcul de $S^{t_i}_{X_i} = (\sigma_{i_1}, \varphi_{i_1}) \oplus \ldots \oplus (\sigma_{i_m}, \varphi_{i_m})$, et d'après l'hypothèse d'induction, $\forall i \in [1, n]$ et $\forall j \in [1, m]$ on a $t_i\sigma_{i_j} \xrightarrow{\varphi_{i_j}}{}^*_{\mathcal{A}_S} X_i$ (3).

Mais le couple passé en entrée est ici le couple $(t_i \trianglelefteq X_i, \top \wedge (X = Y))$ d'après (2). Le résultat, pour chaque t_i, sera alors la disjonction $\oplus$ des couples $(\sigma_{i_j}, \varphi_{i_j} \wedge (X = Y))$,

pour tout $j \in [1, m]$. D'après ce résultat, la décomposition obtenue en (2) est donc de la forme suivante :

$$(\bigoplus_{\forall j \in [1,m]}(\sigma_{1_j}, \varphi_{1_j} \wedge \mathbf{(X{=}Y)})) \otimes \ldots \otimes (\bigoplus_{\forall j \in [1,m]}(\sigma_{n_j}, \varphi_{n_j} \wedge \mathbf{(X{=}Y)})).$$

D'après la 5e règle de normalisation, ceci est distribué de cette manière :

$$\begin{array}{ll}
 & (\sigma_{1_1}, \varphi_{1_1} \wedge (X = Y)) \otimes (\sigma_{2_1}, \varphi_{2_1} \wedge (X = Y)) \otimes \ldots \otimes (\sigma_{n_1}, \varphi_{n_1} \wedge (X = Y)) \\
\oplus & (\sigma_{1_1}, \varphi_{1_1} \wedge (X = Y)) \otimes (\sigma_{2_2}, \varphi_{2_2} \wedge (X = Y)) \otimes \ldots \otimes (\sigma_{n_1}, \varphi_{n_1} \wedge (X = Y)) \\
\oplus & \ldots \\
\oplus & (\sigma_{1_1}, \varphi_{1_1} \wedge (X = Y)) \otimes (\sigma_{2_2}, \varphi_{2_2} \wedge (X = Y)) \otimes \ldots \otimes (\sigma_{n_2}, \varphi_{n_2} \wedge (X = Y)) \\
\oplus & \ldots \\
\oplus & (\sigma_{1_m}, \varphi_{1_m} \wedge (X = Y)) \otimes (\sigma_{2_m}, \varphi_{2_m} \wedge (X = Y)) \otimes \ldots \otimes (\sigma_{n_m}, \varphi_{n_m} \wedge (X = Y))
\end{array}$$

Puis d'après la 1e règle de normalisation, on a alors un ensemble de substitutions appartenant à S_X^t :

$\{(\sigma_{1_1} \cup \sigma_{2_1} \cup \ldots \cup \sigma_{n_1}, \varphi_{1_1} \wedge \varphi_{2_1} \wedge \ldots \wedge \varphi_{n_1} \wedge (X = Y)),$
$\{(\sigma_{1_1} \cup \sigma_{2_2} \cup \ldots \cup \sigma_{n_1}, \varphi_{1_1} \wedge \varphi_{2_2} \wedge \ldots \wedge \varphi_{n_1} \wedge (X = Y)),$
$\ldots,$
$\{(\sigma_{1_1} \cup \sigma_{2_2} \cup \ldots \cup \sigma_{n_2}, \varphi_{1_1} \wedge \varphi_{2_2} \wedge \ldots \wedge \varphi_{n_2} \wedge (X = Y)),$
$\ldots,$
$\{(\sigma_{1_m} \cup \sigma_{2_m} \cup \ldots \cup \sigma_{n_m}, \varphi_{1_m} \wedge \varphi_{2_m} \wedge \ldots \wedge \varphi_{n_m} \wedge (X = Y))\}$

Donc, pour toute substitution $(\sigma, \varphi) \in S_X^t$, on a $(\sigma, \varphi) = (\sigma_1 \cup \ldots \cup \sigma_n, \varphi_1 \wedge \ldots \wedge \varphi_n \wedge \mathbf{(X{=}Y)})$ (4), avec (σ_i, φ_i) une des substitutions de $S_{X_i}^{t_i}$ pour tout $i \in [1, n]$. Or, étant donné que $t = f(t_1, \ldots, t_n)$ est linéaire, et que pour une substitution σ calculée par S_X^t on a $\sigma = \sigma_1 \cup \ldots \cup \sigma_n$, avec σ_i une substitution de $S_{X_i}^{t_i}$, on a donc $t\sigma = f(t_1\sigma_1, \ldots, t_n\sigma_n)$ (5). D'après (3), pour tout $i \in [1, n]$ on a $t_i\sigma_i \xrightarrow{va_i}{}^*_{\mathcal{A}_S} X_i$ si $(\sigma_i, \varphi_i) \in S_{X_i}^{t_i}$. Donc depuis (3), (5) et la définition 4.3.4, on a alors $t\sigma \xrightarrow{\varphi_1 \wedge \varphi_n}{}^*_{\mathcal{A}_S} f(X_1, \ldots, X_n)$ (6). Donc depuis (1) ($f(X_1, \ldots, X_n) \to Y \in \Delta$), (6) et la définition 4.3.4, on a $t\sigma \xrightarrow{\varphi_1 \wedge \varphi_n \wedge (X=Y)}{}^*_{\mathcal{A}_S} X$, soit $t\sigma \xrightarrow{\varphi}{}^*_{\mathcal{A}_S} X$ d'après (4).

□

4.5 À la recherche d'un automate point-fixe concluant

Rappelons que le *Graal* de la complétion d'automates d'arbres est de trouver un automate point-fixe concluant, *i.e.* respectant les propriétés à vérifier (autrement dit, ne comprenant pas de termes interdits). Formellement, étant donné un ensemble de termes Bad, un système de réécriture $\mathcal{R}$ et un automate d'arbres initial $\mathcal{A}$, un automate point-fixe concluant est un automate d'arbres $\mathcal{A}^\star$ tel que $\mathcal{L}(\mathcal{A}^\star)$ soit $\mathcal{R}$-clos (soit $\mathcal{L}(\mathcal{A}^* - \supseteq \mathcal{R}^*(\mathcal{L}(\mathcal{A}))$) et tel que $\mathcal{L}(\mathcal{A}^\star) \cap Bad = \emptyset$. Nous rappelons également que la complétion d'automates d'arbres est correcte uniquement pour les systèmes de réécriture linéaires gauches, ce pourquoi nous considèrerons dans la suite uniquement ce type de système de réécriture.

Dans cette section, pour un STA $\mathcal{A}_S$ donné, un automate d'arbres $\mathcal{A}$, un système de réécriture $\mathcal{R}$ et un ensemble de termes interdits Bad, nous proposons deux formules $\phi^{FP}_{\mathcal{R},\mathcal{A}_S}$ et $\phi^{Bad}_{\mathcal{A}_S}$ telles que chaque instanciation $\mathcal{I}$ de $\mathcal{A}_S$ satisfaisant ces deux formules conduit à un automate point-fixe concluant. De plus, nous définissons une notion de

compatibilité entre l'automate $\mathcal{A}$ de départ et un STA $\mathcal{A}_S$ assurant que l'automate $\mathcal{A}_S^{\mathcal{I}}$ est un automate point-fixe concluant vis-à-vis de $\mathcal{A}$, ce qui implique, nous le verrons, que son langage contient celui de $\mathcal{A}$.

4.5.1 Formule point-fixe

La contrainte présentée ci-dessous dépeint une condition, construite à partir de $\mathcal{A}_S$, à satisfaire pour chaque instance $\mathcal{I}$, pour assurer que l'automate $\mathcal{A}_S^{\mathcal{I}}$ est $\mathcal{R}$-clos. Dans [Feuillade et al., 2004], un automate d'arbres $\mathcal{A}$ est $\mathcal{R}$-clos si $\forall l \to r \in \mathcal{R}$, $\forall \sigma : \mathcal{X} \mapsto \mathcal{Q}$ et $\forall q$, si $l\sigma \to_{\mathcal{A}}^* q$ alors $r\sigma \to_{\mathcal{A}}^* q$. Pour un STA, cela ce traduit par $\forall l \to r \in \mathcal{R}$, $\forall X \in \mathcal{X}_{\mathcal{Q}}$, $\forall(\sigma, \varphi_1) \in S_X^l$, si $l\sigma \xrightarrow{\varphi_1}{}^*_{\mathcal{A}_S} X$ alors $r\sigma \xrightarrow{\varphi_2}{}^*_{\mathcal{A}_S} X$, avec φ_2 tel que $(_, \varphi_2) \in S_X^{r\sigma}$ soit φ_2 représentant l'ensemble des conditions de réduction de $r\sigma$ dans l'automate. Pour résumer grossièrement le fonctionnement de la contrainte point-fixe, si la formule va_1 est respectée (*i.e.* si $l\sigma$ se réduit dans l'automate), alors la formule va_2 doit l'être également (*i.e.* $r\sigma$ doit également être reconnu par l'automate). Notons qu'ici, on cherche à réduire les parties droite et gauche de chaque règle de réécriture sur le même état symbolique X (si $l\sigma \xrightarrow{\varphi_1}{}^*_{\mathcal{A}_S} X$ alors $r\sigma \xrightarrow{\varphi_2}{}^*_{\mathcal{A}_S} X$). On pourrait également dire : si $l\sigma \xrightarrow{\varphi_1}{}^*_{\mathcal{A}_S} X$ alors $r\sigma \xrightarrow{\varphi_2}{}^*_{\mathcal{A}_S} Y$ et $X = Y$ mais ceci n'ajoute pas de possibilités supplémentaires et ne fait qu'ajouter une conjonction de plus à la contrainte point-fixe. Effectivement, si $r\sigma$ ne se réduit pas directement en X, le calcul de substitution ajoutera elle-même l'égalité nécessaire pour permettre la réduction en X. Par exemple, si $l\sigma \xrightarrow{\varphi_1}{}^*_{\mathcal{A}_S} X$, que $r\sigma = f(X_1)$ et que l'on a la transition $f(X_1) \to X_2$ alors $f(X_1) \xrightarrow{X_2=X}{}^*_{\mathcal{A}_S} X$, ou bien $f(X_1) \xrightarrow{X_2=Y}{}^*_{\mathcal{A}_S} Y$ et $Y = X$, soit deux formules équivalentes, la deuxième étant inutilement plus complexe.

DÉFINITION 4.5.1 (Formule point-fixe : $\phi^{FP}_{\mathcal{R},\mathcal{A}_S}$)
Soient $\mathcal{A}_S$ un STA $\langle \mathcal{F}, \mathcal{X}_{\mathcal{Q}}, \mathcal{X}_{\mathcal{Q}}^f, \Delta \rangle$ et $\mathcal{R}$ un système de réécriture linéaire gauche. On note $\phi^{FP}_{\mathcal{R},\mathcal{A}_S}$ la formule définie comme suit :

$$\phi^{FP}_{\mathcal{R},\mathcal{A}_S} \stackrel{def}{=} \bigwedge_{l \to r \in \mathcal{R}} \bigwedge_{X \in \mathcal{X}_{\mathcal{Q}}} \bigwedge_{(\sigma,\varphi_1) \in S_X^l} (\varphi_1 \Rightarrow \bigvee_{(_,\varphi_2) \in S_X^{r\sigma}} \varphi_2)$$

où S_X^l et $S_X^{r\sigma}$ sont respectivement les ensembles de solutions de l'algorithme de filtrage pour $l \trianglelefteq X$, et $r\sigma \trianglelefteq X$ (utilisant la règle (État Symbolique)).

EXEMPLE *4.11*
Soient $\mathcal{A}_{S_1}$ le STA de l'exemple 4.9 et $\mathcal{R} = \{s(x) \to s(s(x))\}$ un système de réécriture. On a alors : $\phi^{FP}_{\mathcal{R},\mathcal{A}_S} = (\top \Rightarrow X_0 = X_1) \wedge (X_0 = X_1 \Rightarrow X_0 = X_1)$.

En effet, si l'on résout le problème de filtrage $s(x) \trianglelefteq X$ pour tout $X \in \mathcal{X}_{\mathcal{Q}}$, alors on doit résoudre $s(x) \trianglelefteq X_0$ et $s(x) \trianglelefteq X_1$. On a alors :

$$\text{(Configuration)} \quad \frac{(s(x) \trianglelefteq X_0, \top)}{(s(x) \trianglelefteq s(X_0), X_0 = X_1)} \quad (s(X_0) \to X_1 \in \Delta)$$

$$\text{(Décomposition)} \quad \frac{}{(x \trianglelefteq X_0, X_0 = X_1)}$$

$$\text{(Variable)} \quad \frac{}{\{(x \mapsto X_0, X_0 = X_1)\}} \quad (x \in \mathcal{X})$$

et

$$\begin{array}{rcl}
\text{(Configuration)} & \dfrac{(s(x) \trianglelefteq X_1, \top)}{(s(x) \trianglelefteq s(X_0), \top)} & (s(X_0) \to X_1 \in \Delta) \\
\text{(Décomposition)} & \dfrac{}{(x \trianglelefteq X_0, \top)} & \\
\text{(Variable)} & \dfrac{}{\{(x \mapsto X_0, \top)\}} & (x \in \mathcal{X})
\end{array}$$

Pour chaque état X de $\mathcal{X}_{\mathcal{Q}}$, et pour chaque substitution $(\sigma, \varphi_1) \in S_X^l$ calculée, on doit ensuite calculer les substitutions $(_, \varphi_2)$ du problème de matching $r\sigma \trianglelefteq X$. Ici, on doit donc résoudre $s(s(X_0)) \trianglelefteq X_0$ et $s(s(X_0)) \trianglelefteq X_1$. On a donc :

$$\begin{array}{rcl}
\text{(Configuration)} & \dfrac{(s(s(X_0)) \trianglelefteq X_0, \top)}{(s(s(X_0)) \trianglelefteq s(X_0), X_0 = X_1)} & (s(X_0) \to X_1 \in \Delta) \\
\text{(Décomposition)} & \dfrac{}{(s(X_0) \trianglelefteq X_0, X_0 = X_1)} & \\
\text{(Configuration)} & \dfrac{}{(s(X_0) \trianglelefteq s(X_0), X_0 = X_1)} & (s(X_0) \to X_1 \in \Delta) \\
\text{(Décomposition)} & \dfrac{}{(X_0 \trianglelefteq X_0, X_0 = X_1)} & \\
\text{(État Symbolique)} & \dfrac{}{(\emptyset, X_0 = X_1)} & (X_0 \in \mathcal{X}_{\mathcal{Q}})
\end{array}$$

et

$$\begin{array}{rcl}
\text{(Configuration)} & \dfrac{(s(s(X_0)) \trianglelefteq X_1, \top)}{(s(s(X_0)) \trianglelefteq s(X_0), \top)} & (s(X_0) \to X_1 \in \Delta) \\
\text{(Décomposition)} & \dfrac{}{(s(X_0) \trianglelefteq X_0, \top)} & \\
\text{(Configuration)} & \dfrac{}{(s(X_0) \trianglelefteq s(X_0), X_0 = X_1)} & (s(X_0) \to X_1 \in \Delta) \\
\text{(Décomposition)} & \dfrac{}{(X_0 \trianglelefteq X_0, X_0 = X_1)} & \\
\text{(État Symbolique)} & \dfrac{}{\{(\emptyset, X_0 = X_1)\}} & (X_0 \in \mathcal{X}_{\mathcal{Q}})
\end{array}$$

Résumons : on a alors $S_{X_0}^{s(x)} = (x \trianglelefteq X_0, X_0 = X_1)$, $S_{X_1}^{s(x)} = (x \trianglelefteq X_0, \top)$, $S_{X_0}^{s(s(X_0))} = (\emptyset, X_0 = X_1)$ et $S_{X_0}^{s(s(X_1))} = (\emptyset, X_0 = X_1)$. En appliquant la définition 4.5.1, on obtient :

$$\phi_{\mathcal{R},\mathcal{A}_S}^{FP} = (\top \Rightarrow X_0 = X_1) \wedge (X_0 = X_1 \Rightarrow X_0 = X_1).$$

L'instance $\mathcal{A}_{S_1}^{\mathcal{I}_1}$ de l'exemple 4.5 satisfait cette formule. ◀

EXEMPLE *4.12*
Soient $\mathcal{A}_{S_2}$ le STA de l'exemple 4.10 et $\mathcal{R} = \{cons(x, nil) \to cons(r, cons(x, nil))\}$ un système de réécriture.

De la même manière que pour l'exemple 4.11, on doit alors résoudre $cons(x, nil) \trianglelefteq X$ pour tout $X \in \mathcal{X}_{\mathcal{Q}}$.

On a alors :

$$
\begin{array}{rl}
\text{(Conf.)} & \dfrac{(cons(x,nil) \trianglelefteq X, \top)}{(cons(x,nil) \trianglelefteq cons(X_r, X_{nil}), X = X_0) \oplus (cons(x,nil) \trianglelefteq cons(X_p, X_0), X = X_1)} \\
\text{(Décomp.)} & \dfrac{((x \trianglelefteq X_r, X = X_0) \otimes (nil \trianglelefteq X_{nil}, X = X_0)) \oplus ((x \trianglelefteq X_p, X = X_1) \otimes (nil \trianglelefteq X_0, X = X_1))}{(\{x \mapsto X_r\}, X = X_0) \otimes (\emptyset, X = X_0)) \oplus ((\{x \mapsto X_p\}, X = X_1) \otimes (\emptyset, X = X_1 \wedge X_0 = X_{nil}))} \\
\text{(Var.), (Const.)} & \\
\text{(Norm.)} & \dfrac{}{\{(\{x \mapsto X_r\}, X = X_0), (\{x \mapsto X_p\}, X = X_1 \wedge X_0 = X_{nil})\}}
\end{array}
$$

On a donc $S_{X_p}^{cons(x,nil)} = \{(\{x \mapsto X_r\}, X_p = X_0), (\{x \mapsto X_p\}, (X_p = X_1) \wedge (X_0 = X_{nil}))\}$, $S_{X_r}^{cons(x,nil)} = \{(\{x \mapsto X_r\}, X_r = X_0), (\{x \mapsto X_p\}, (X_r = X_1) \wedge (X_0 = X_{nil}))\}$, etc.

On doit ensuite résoudre, pour tout $X \in \mathcal{X}_\mathcal{Q}$ et pour tout $(\sigma, \varphi) \in S_X^{cons(x,nil)}$, le problème de filtrage $r\sigma \trianglelefteq X$. Donc ici, pour tout $x \in \mathcal{X}_\mathcal{Q}$, $cons(r, cons(X_r, nil)) \trianglelefteq X$ et $cons(r, cons(X_p, nil)) \trianglelefteq X$. Sur l'exemple et 4.10 la figure 4.7, nous avions calculé $cons(r, cons(x, nil)) \trianglelefteq X_0$. En remplaçant X_0 par chaque X de $\mathcal{X}_\mathcal{Q}$ et x par X_r ou X_p, nous avons alors :

$$
\begin{array}{rcl}
S_X^{cons(r,cons(X_r,nil))} & = & (\emptyset, ((X_0 = X_{nil}) \wedge (X_0 = X)) \vee ((X_p = X_r) \wedge (X_1 = X))) \\
& & \oplus(\emptyset, ((X_p = X_r) \wedge (X_1 = X_{nil}) \wedge (X_0 = X_{nil}) \wedge (X_0 = X)) \\
& & \vee((X_p = X_r) \wedge (X_0 = X_{nil}) \wedge (X_0 = X_1) \wedge (X_1 = X))) \\
& = & (\emptyset, ((X_0 = X_{nil}) \wedge (X_0 = X)) \vee ((X_p = X_r) \wedge (X_1 = X)) \\
& & \vee((X_p = X_r) \wedge (X_1 = X_{nil}) \wedge (X_0 = X_{nil}) \wedge (X_0 = X)) \\
& & \vee((X_p = X_r) \wedge (X_0 = X_{nil}) \wedge (X_0 = X_1) \wedge (X_1 = X))) \\
& = & (\emptyset, ((X_0 = X_{nil}) \wedge (X_0 = X)) \vee ((X_p = X_r) \wedge (X_1 = X))),
\end{array}
$$

et

$$
\begin{array}{rcl}
S_X^{cons(r,cons(X_p,nil))} & = & (\emptyset, ((X_p = X_r) \wedge (X_0 = X_{nil}) \wedge (X_0 = X)) \vee ((X_p = X_r) \\
& & \wedge(X_1 = X)) \vee ((X_1 = X_{nil}) \wedge (X_0 = X_{nil}) \wedge (X_0 = X)) \\
& & \vee((X_p = X_r) \wedge (X_0 = X_{nil}) \wedge (X_0 = X_1) \wedge (X_1 = X))) \\
& = & (\emptyset, ((X_p = X_r) \wedge (X_0 = X_{nil}) \wedge (X_0 = X)) \vee ((X_p = X_r) \\
& & \wedge(X_1 = X)) \vee ((X_1 = X_{nil}) \wedge (X_0 = X_{nil}) \wedge (X_0 = X))).
\end{array}
$$

On a alors :

$$
\begin{array}{rcl}
\phi_{\mathcal{R},\mathcal{A}_S}^{FP} & = & \bigwedge_{\forall X \in \mathcal{X}_\mathcal{Q}}((X = X_0) \Rightarrow (((X_0 = X_{nil}) \wedge (X_0 = X)) \\
& & \vee((X_p = X_r) \wedge (X_1 = X)))) \\
& & \wedge(((X = X_1) \wedge (X_0 = X_{nil})) \Rightarrow (((X_p = X_r) \wedge (X_0 = X_{nil}) \wedge (X_0 = X)) \\
& & \vee((X_p = X_r) \wedge (X_1 = X)) \vee ((X_1 = X_{nil}) \wedge (X_0 = X_{nil}) \wedge (X_0 = X)))).
\end{array}
$$

Une instance de $\mathcal{A}_{S_2}$ pour laquelle $X_0 = X_1 = X_{nil}$, ou $X_p = X_r$ et $X_0 = X_{il}$, ou $X_p = X_r$ et $X_0 = X_1$ est un point-fixe. L'automate $\mathcal{A}_{S_2}^{I_1}$ de l'exemple 4.6 et de la figure 4.4 satisfait la dernière conjonction, c'est donc un point-fixe. ◀

Nous établissons formellement l'utilité de la formule $\phi_{\mathcal{R},\mathcal{A}_S}^{FP}$ dans la proposition suivante. En effet, elle permet de caractériser l'ensemble des instances point-fixes du STA $\mathcal{A}_S$. En d'autres mots, toute instanciation $\mathcal{I}$ de $\mathcal{A}_S$ respectant cette formule conduit à un point-fixe.

PROPOSITION 4.5.2
Soient $\mathcal{A}_S$ un STA et $\mathcal{R}$ un système de réécriture linéaire gauche. Soient $\mathcal{Q}$ un ensemble d'états et $\mathcal{I}$ une instanciation $\mathcal{X}_\mathcal{Q} \to \mathcal{Q}$. Alors, $\mathcal{I} \models \phi_{\mathcal{R},\mathcal{A}_S}^{FP}$ si et seulement si $\mathcal{A}_S^\mathcal{I}$ est $\mathcal{R}$-clos.

Pour prouver cette proposition, nous avons tout d'abord besoin de prouver le lemme suivant.

LEMME 4.5.3
Soient $t \in \mathcal{T}(\mathcal{F}, \mathcal{X})$ un terme, $\mathcal{A}_S = \langle \mathcal{F}, \mathcal{X}_\mathcal{Q}, \mathcal{X}^f_\mathcal{Q}, \Delta \rangle$ un STA, et X un état symbolique de $\mathcal{X}_\mathcal{Q}$. Soient S^t_X l'ensemble de solutions du problème de filtrage $t \trianglelefteq X$, σ une substitution telle que $\sigma : \mathcal{X} \mapsto \mathcal{X}_\mathcal{Q}$ et $S^{t\sigma}_X$ l'ensemble des solutions du problème de filtrage $t\sigma \trianglelefteq X$. Soit $\mu : \mathcal{X} \mapsto \mathcal{X}_\mathcal{Q}$ une substitution, φ_1 et φ_2 des formules de $\rho(\mathcal{X}_\mathcal{Q})$, on a alors : pour chaque $(\emptyset, \varphi_2) \in S^{t\sigma}_X$, il existe $(\mu, \varphi_1) \in S^t_X$ tel que

$$\varphi_2 = \varphi_1 \wedge \bigwedge_{x \in \mathcal{V}ar(t)} (\mu(x) = \sigma(x)).$$

Démonstration. Ce lemme est une conséquence de la règle (État Symbolique) de la définition 4.4.1. En effet, soit t un terme de $\mathcal{T}(\mathcal{F}, \mathcal{X})$ tel que $t = f(s_1, \ldots, s_n)$. Étant donné que $t \in \mathcal{T}(\mathcal{F}, \mathcal{X})$, on a $f(s_1, \ldots, s_n) \overset{\varphi}{\rightarrow}^* f(x_1, \ldots, x_n)$ avec $\varphi \in \rho(\mathcal{X}_\mathcal{Q})$ calculé lors de l'algorithme de filtrage jusqu'à cette étape, et où $\forall i \in [1, n]$, x_i est soit une constante, soit une variable de $\mathcal{X}$.

L'algorithme de filtrage calculant S^t_X termine donc nécessairement par des couples $(x_1 \trianglelefteq X_1, \varphi), \ldots, (x_n \trianglelefteq X_n, \varphi)$, sur lesquels il faut appliquer soit la règle de déduction (Constante), soit la règle de déduction (Variable). Soit, pour tout $i \in [1, n]$, on a le couple résultat $(\{x_i \mapsto X_i\}, \varphi)$ ou $(\emptyset, \varphi \wedge (X_i = Y_1) \wedge \ldots \wedge (X_i = Y_k))$ si pour $j \in [1, k]$, $x_i \rightarrow Y_j$ est une transition de l'automate. On a alors :

$$S^t_X = (\mu, \varphi_1) = (\{x_i \trianglelefteq X_i \mid \forall x_i \in \mathcal{V}ar(t)\}, \varphi \wedge \bigwedge_{\forall i \ t.q. \ x_i \in \mathcal{F}^0} ((X_i = Y_1) \wedge \ldots \wedge (X_i = Y_k)))(*).$$

Soit $\sigma : \mathcal{X} \mapsto \mathcal{X}_\mathcal{Q}$ une substitution telle que $\sigma = \{x_i \trianglelefteq Z_i \mid \forall x_i \in \mathcal{V}ar(t)\}$. Alors $t\sigma = f(s_1\sigma, \ldots, s_n\sigma) \overset{\varphi}{\rightarrow}^* f(\sigma(x_1), \ldots, \sigma(x_n))$. L'algorithme de filtrage calculant $S^{t\sigma}_X$ termine donc nécessairement par des couples $(\sigma(x_1) \trianglelefteq X_1, \varphi), \ldots, (\sigma(x_n) \trianglelefteq X_n, \varphi)$. On retrouve les mêmes $X_1, \ldots, X_n$ que pour le calcul de S^t_X, ainsi que φ, car jusqu'à cette étape, les règles s'appliquent de la même manière. Pour tout $i \in [1, n]$, si x_i est une constante, alors $\sigma(x_i) = x_i$ et l'algorithme de filtrage calculant $S^{t\sigma}_X$ retournera la même chose que pour S^t_X, *i.e.* $(\emptyset, \varphi \wedge (X_i = Y_1) \wedge \ldots \wedge (X_i = Y_k))$.

Si $x_i \in Var(t)$, alors $\sigma(x_i) = Z_i$ et on a le couple $(Z_i \trianglelefteq X_i, \varphi)$ sur lequel il faut appliquer la règle (État Symbolique). On obtient alors le couple $(\emptyset, \varphi \wedge (X_i = Z_i))$.

On a alors $S^{t\sigma}_X = (\emptyset, \varphi_2) = (\emptyset, \varphi \wedge \bigwedge_{\forall i \ t.q. \ x_i \in \mathcal{F}^0}((X_i = Y_1) \wedge \ldots \wedge (X_i = Y_k)) \wedge \bigwedge_{\forall x_i \in \mathcal{V}ar(t)}(X_i = Z_i))$ soit, d'après (*),

$$(\emptyset, \varphi_1 \wedge \bigwedge_{\forall x_i \in \mathcal{V}ar(t)} (X_i = Z_i)).$$

Or $X_i = \mu(x_i)$ et $Z_i = \sigma(x_i)$ si $x_i \in \mathcal{V}ar(t)$. On a donc bien $\varphi_2 = \varphi_1 \wedge \bigwedge_{x \in \mathcal{V}ar(t)}(\mu(x) = \sigma(x))$.

□

Montrons désormais la propriété 4.5.2.

Démonstration. Montrons tout d'abord que si $\mathcal{I} \models \phi^{FP}_{\mathcal{R},\mathcal{A}_S}$ alors $\mathcal{A}^{\mathcal{I}}_S$ est $\mathcal{R}-$ clos. Selon la définition 4.5.1, on a : $\phi^{FP}_{\mathcal{R},\mathcal{A}_S} = \bigwedge_{l\to r\in\mathcal{R}}(\bigwedge_{\forall X\in\mathcal{X}_\mathcal{Q},(\sigma,\varphi_1)\in S^l_X}(\varphi_1 \Rightarrow \bigvee_{\forall(_,\varphi_2)\in S^{r\sigma}_X}(\varphi_2)))$, où S^l_X et $S^{r\sigma}_X$ sont respectivement les ensembles de solutions des problèmes de filtrage $l \trianglelefteq X$ et $r\sigma \trianglelefteq X$. Selon la définition 4.2.2, $\mathcal{I} \models \bigwedge_{l\to r\in\mathcal{R}}(\bigwedge_{\forall X\in\mathcal{X}_\mathcal{Q},(\sigma,\varphi_1)\in S^l_X}(\varphi_1 \Rightarrow \bigvee_{\forall(_,\varphi_2)\in S^{r\sigma}_X}(\varphi_2)))$ peut être réécrit en : $\forall l \to r \in \mathcal{R}$, $\forall X \in \mathcal{X}_\mathcal{Q}$ et $\forall(\sigma,\varphi_1,) \in S^l_X$, $\mathcal{I} \models (\varphi_1 \Rightarrow \bigvee_{\forall(_,\varphi_2)\in S^{r\sigma}_X}(\varphi_2)))$. Selon la définition de $\models$, on peut alors déduire que : $\mathcal{I} \models (\varphi_1 \Rightarrow \bigvee_{\forall(_,\varphi_2)\in S^{r\sigma}_X}(\varphi_2)))$ si et seulement si $\mathcal{I} \models \neg\varphi_1$ ou $\mathcal{I} \models \bigvee_{\forall(_,\varphi_2,)\in S^{r\sigma}_X}(\varphi_2)))$. Procédons par analyse au cas par cas :

- $\mathcal{I} \models \varphi_1$ et $\mathcal{I} \models \bigvee_{\forall(_,\varphi_2)\in S^{r\sigma}_X}(\varphi_2)))$: rappelons que $(\sigma,\varphi_1) \in S^l_X$. Selon la proposition 4.4.3, on peut alors déduire que $l\sigma \xrightarrow{\varphi_1}{}^*_{\mathcal{A}_S} X$. Par conséquent, si $\mathcal{I} \models \varphi_1$ alors $\mathcal{I} \models \bigvee_{\{t \xrightarrow{\varphi'_1}{}^*_{\mathcal{A}_S} X\}} \varphi'_1 = \texttt{Reco}(t,X)$. En appliquant la proposition 4.3.5, On peut alors déduire que :

$$l\sigma \circ \mathcal{I} \to^*_{\mathcal{A}^{\mathcal{I}}_S} \mathcal{I}(X). \tag{4.4}$$

  Par hypothèse, $\mathcal{I} \models \bigvee_{\forall(_,\varphi_2)\in S^{r\sigma}_X}(\varphi_2)))$. Par conséquent, il existe $(_,\gamma) \in S^{r\sigma}_X$ tel que $\mathcal{I} \models \gamma$. Selon le lemme 4.5.3, on peut déduire qu'il existe $(\sigma',\varphi'_1) \in S^r_X$ tel que $\gamma = \varphi'_1 \wedge \bigwedge_{x\in\mathcal{V}ar(r)}(\sigma(x) = \sigma'(x))$. De plus, on peut également déduire que $\mathcal{I} \models \varphi'_1 \wedge \bigwedge_{x\in\mathcal{V}ar(r)}(\sigma(x) = \sigma'(x))$. Selon la proposition 4.4.3, $r\sigma' \circ \mathcal{I} \to^*_{\mathcal{A}^{\mathcal{I}}_S} \mathcal{I}(X)$. Comme on a $\mathcal{I} \models \varphi'_1 \wedge \bigwedge_{x\in\mathcal{V}ar(r)}(\sigma(x) = \sigma'(x))$, cela implique alors que $r\sigma \circ \mathcal{I} = r\sigma' \circ \mathcal{I}$. Donc,

$$r\sigma \circ \mathcal{I} \to^*_{\mathcal{A}^{\mathcal{I}}_S} \mathcal{I}(X). \tag{4.5}$$

  Alors, si l'on a (4.4) alors on a également (4.5) permettant de caractériser un automate $\mathcal{R}-$clos.
- $\mathcal{I} \not\models \varphi_1$: comme φ_1 représente une réduction particulière de l à l'état symbolique X, $\mathcal{I} \models \neg\varphi_1$ signifie que la réduction impliquée ne peut pas s'appliquer dans l'automate d'arbres $\mathcal{A}^{\mathcal{I}}_S$. Étant donné que cette réduction n'est pas possible, il n'est nul besoin de vérifier si $r\sigma \circ \mathcal{I} \to^*_{\mathcal{A}^{\mathcal{I}}_S} \mathcal{I}(X)$ ou non, dans le but de garantir que $\mathcal{A}^{\mathcal{I}}_S$ est $\mathcal{R}-$clos.

Maintenant, prouvons que si $\mathcal{A}^{\mathcal{I}}_S$ est $\mathcal{R}-$ clos alors $\mathcal{I} \models \phi^{FP}_{\mathcal{R},\mathcal{A}_S}$. Comme vu dans le chapitre 2, $\mathcal{A}^{\mathcal{I}}_S$ est $\mathcal{R}-$clos si pour chaque règle de réécriture $l \to r \in \mathcal{R}$, pour chaque substitution $\mu : \mathcal{X} \mapsto \mathcal{Q}$, si $l\mu$ est reconnu par $\mathcal{A}^{\mathcal{I}}_S$ dans l'état $\mathcal{I}(X)$ alors $r\mu$ l'est aussi. Selon la proposition 4.3.5, on peut alors déduire qu'il existe $\sigma : \mathcal{X} \mapsto \mathcal{X}_\mathcal{Q}$ tel que $\mathcal{I} \models \texttt{Reco}(l\sigma,X)$ et $\mu = \sigma \circ \mathcal{I}$. Comme $\mathcal{I} \models \texttt{Reco}(l\sigma,X)$, on peut alors déduire qu'il existe φ_1 tel que $l\sigma \xrightarrow{\varphi_1}{}^*_{\mathcal{A}_S} X$ et $\mathcal{I} \models \varphi_1$. Comme $r\mu$ est reconnu par $\mathcal{A}^{\mathcal{I}}_S$ dans l'état $\mathcal{I}(X)$, alors d'après la proposition 4.3.5, on déduit qu'il existe $\sigma' : \mathcal{X} \mapsto \mathcal{X}_\mathcal{Q}$ tel que $\mathcal{I} \models \texttt{Reco}(r\sigma',X)$. Étant donné que $\mathcal{I} \models \texttt{Reco}(r\sigma',X)$, on déduit qu'il existe alors φ_2 tel que $r\sigma' \xrightarrow{\varphi_2}{}^*_{\mathcal{A}_S} X$ et $\mathcal{I} \models \varphi_2$. Notons que $(\sigma',\varphi_2) \in S^r_X$, avec S^r_X l'ensemble de solutions du problème de filtrage $r \trianglelefteq X$. Maintenant, considérons $r\sigma$. D'après le lemme 4.5.3, on peut construire γ tel que $\gamma = \varphi_2 \wedge \bigwedge_{x\in\mathcal{V}ar(r)}(\sigma(x) = \sigma'(x))$ et $(_,\gamma) \in S^{r\sigma}_X$ (considérant $S^{r\sigma}_X$ comme l'ensemble de solutions du problème de filtrage $r\sigma \trianglelefteq X$). Par construction, $\sigma \circ \mathcal{I}(x) = \sigma' \circ \mathcal{I}(x)$, pour chaque $x \in \mathcal{V}ar(r)$. Par conséquent, on a alors $\mathcal{I} \models \gamma$. Alors, $\mathcal{I}$ satisfait également la formule $\bigvee_{\forall(_,\varphi_2)\in S^{r\sigma}_X}(\varphi_2)))$. En itérant ce processus pour chaque

substitution $\sigma : \mathcal{X} \mapsto \mathcal{Q}$, pour chaque règle $l \to r$ et pour chaque état de $\mathcal{A}_S^{\mathcal{I}}$, on obtient alors $\mathcal{I} \models \phi^{FP}_{\mathcal{R},\mathcal{A}_S}$. Ce qui conclue cette preuve. □

4.5.2 Formule de non-reconnaissance des termes interdits

Nous sommes désormais capables, pour un STA $\mathcal{A}_S$ donné, de formaliser une condition point-fixe. Cependant, nous recherchons un point-fixe particulier. En effet, supposons qu'il existe une instanciation $\mathcal{I}$ telle que $\mathcal{I} \models \phi^{FP}_{\mathcal{R},\mathcal{A}_S}$. Rappelons que notre but est de trouver un automate point-fixe $\mathcal{A}^\star$ tel que $\mathcal{L}(\mathcal{A}^\star) \cap Bad = \emptyset$. La prochaine définition propose une formule $\phi^{Bad}_{\mathcal{A}_S}$ caractérisant la non reconnaissance de tous les éléments de l'ensemble Bad par chaque instance $\mathcal{A}_S^{\mathcal{I}}$ telle que $\mathcal{I}$ satisfait également cette formule. Autrement dit, chaque instanciation $\mathcal{I}$ d'un STA $\mathcal{A}_S$ respectant $\phi^{Bad}_{\mathcal{A}_S}$ conduit à un automate $\mathcal{A}_S^{\mathcal{I}}$ ne reconnaissant aucun terme interdit et satisfaisant donc la propriété à vérifier.

DÉFINITION 4.5.4 ($\phi^{Bad}_{\mathcal{A}_S}$)
Soit $\mathcal{A}_S$ un STA $\langle \mathcal{F}, \mathcal{X}_\mathcal{Q}, \mathcal{X}^f_\mathcal{Q}, \Delta\rangle$ et Bad un ensemble fini de termes clos. On note $\phi^{Bad}_{\mathcal{A}_S}$ la formule définie sur $\rho(\mathcal{X}_\mathcal{Q})$ de la manière suivante :

$$\phi^{Bad}_{\mathcal{A}_S} \stackrel{def}{=} \bigwedge_{t \in Bad} \bigwedge_{X \in \mathcal{X}^f_\mathcal{Q}} \bigwedge_{(_,\varphi) \in S^t_X} \neg\varphi.$$

Pour résumer grossièrement le principe de $\phi^{Bad}_{\mathcal{A}_S}$, on va calculer, pour chaque terme interdit $t \in Bad$, toutes les *formules φ conditionnant la réduction de t* dans notre automate, *i.e.* telles que $t \stackrel{\varphi}{\to}^*_{\mathcal{A}_S} X$ pour n'importe quel X. L'ensemble des φ possibles peut être calculé grâce à l'algorithme de filtrage S^t_X. Soit $\mathcal{A}_S$ un STA $\langle \mathcal{F}, \mathcal{X}_\mathcal{Q}, \mathcal{X}^f_\mathcal{Q}, \Delta\rangle$ et Bad un ensemble fini de termes clos, le fait qu'au moins un terme interdit soit reconnu par $\mathcal{A}_S$ s'exprime donc de cette manière :

$$\bigvee_{t \in Bad} \bigvee_{X \in \mathcal{X}^f_\mathcal{Q}} \bigvee_{(_,\varphi) \in S^t_X} \varphi(*)$$

Rappelons qu'un automate vérifiant la propriété n'accepte aucun terme interdits. On veut donc qu'**aucun** de ces termes interdits ne se réduisent dans l'automate. La formule $\phi^{Bad}_{\mathcal{A}_S}$, caractérisant qu'un automate vérifie la propriété, sera donc une négation de la formule $(*)$. La formule $\phi^{Bad}_{\mathcal{A}_S}$ n'est définie que pour un nombre fini de termes interdits, étant donné qu'elle énumère toutes les possibilités que ces derniers soient reconnus par l'automate.

EXEMPLE *4.13*
Rappelons $\mathcal{A}_{S_1}$ le STA de l'exemple 4.9. $\mathcal{A}_{S_1}$ possède l'ensemble de transitions suivant :

$$\begin{aligned}\Delta = \{ \quad & a \to X_0, b \to X_1, \\ & s(X_0) \to X_1 \ \}.\end{aligned}$$

Soit $Bad = \{s(b)\}$. Ici, le seul état final est X_1. On doit alors calculer $S^{s(b)}_{X_1}$.

$$\text{(Configuration)}\quad \frac{(s(b) \trianglelefteq X_1, \top)}{(s(b) \trianglelefteq s(X_0), \top)} \quad (s(X_0) \to X_1 \in \Delta)$$

$$\text{(Décomposition)}\quad \frac{}{(b \trianglelefteq X_0, \top)}$$

$$\text{(Constante)}\quad \frac{}{(\emptyset, X_0 = X_1)} \quad (b \to X_1 \in \Delta)$$

On a alors : $\phi^{Bad}_{\mathcal{A}_{S_1}} = (X_0 \neq X_1)$. L'instance $\mathcal{A}^{\mathcal{I}_2}_{S_1}$ de l'exemple 4.5 satisfait cette formule.

◀

EXEMPLE *4.14*
Rappelons $\mathcal{A}_{S_2}$ le STA de l'exemple 4.10. $\mathcal{A}_{S_2}$ possède l'ensemble de transitions suivant :

$$\begin{aligned}\Delta = \{ \quad & p \to X_p, r \to X_r, nil \to X_{nil},\\ & cons(X_r, X_{nil}) \to X_0,\\ & cons(X_p, X_0) \to X_1 \ \}.\end{aligned}$$

Si l'on observe le système de réécriture $\mathcal{R} = \{cons(x, nil) \to cons(r, cons(x, nil))\}$ et l'automate initial reconnaissant le terme $cons(p, cons(r, nil))$, on peut alors remarquer que le langage qui représente normalement l'ensemble des termes accessibles est $cons(p, [cons(r,]^* \ nil)[)]^*)$. Une idée serait donc de vérifier qu'il n'existe pas de termes commençant par r. Soit $Bad = \{cons(r, cons(r, nil))\}$. Ici, le seul état final est X_1. On doit alors calculer $S^{cons(r,cons(r,nil))}_{X_1}$.

On a alors :

$$\begin{aligned}S^{cons(r,cons(r,nil))}_{X_1} \quad = \quad & \{(\emptyset, ((X_0 = X_1) \wedge (X_0 = X_{nil})) \vee (X_p = X_r)\\ & \vee((X_0 = X_1) \wedge (X_0 = X_{nil}) \wedge (X_1 = X_{nil}) \wedge (X_p = X_r))\\ & \vee((X_0 = X_{nil}) \wedge (X_0 = X_1) \wedge (X_p = X_r))\}\\ = \quad & \{(\emptyset, ((X_0 = X_1) \wedge (X_0 = X_{nil})) \vee (X_p = X_r))\}.\end{aligned}$$

On a alors $\phi^{Bad}_{\mathcal{A}_{S_2}} = ((X_0 \neq X_1) \vee (X_0 \neq X_{nil})) \wedge (X_p \neq X_r))$. L'instance $\mathcal{A}^{\mathcal{I}_2}_{S_2}$ de l'exemple 4.6 satisfait cette formule. ◀

La proposition suivante permet de montrer qu'une instanciation $\mathcal{I}$ satisfaisant la formule $\phi^{Bad}_{\mathcal{A}_S}$ conduit à un automate n'acceptant aucun terme interdit.

PROPOSITION 4.5.5
Soient $\mathcal{A}_S = \langle \mathcal{F}, \mathcal{X}_\mathcal{Q}, \mathcal{X}^f_\mathcal{Q}, \Delta \rangle$ un STA, et Bad un ensemble fini de termes clos. Soient $\mathcal{Q}$ un ensemble d'états et $\mathcal{I} : \mathcal{X}_\mathcal{Q} \mapsto \mathcal{Q}$ une instanciation. Alors, $\mathcal{I} \models \phi^{Bad}_{\mathcal{A}_S}$ si et seulement si $\mathcal{L}(\mathcal{A}^\mathcal{I}_S) \cap Bad = \emptyset$.

Démonstration. D'après la définition 4.5.4, $\phi^{Bad}_{\mathcal{A}_S} = \bigwedge_{\forall t \in Bad}(\bigwedge_{\forall X \in \mathcal{X}^f_\mathcal{Q}, (_,\varphi) \in S^t_X} \neg\varphi)$.
Donc $\mathcal{I} \models \phi^{Bad}_{\mathcal{A}_S} \Leftrightarrow \mathcal{I} \models \bigwedge_{\forall t \in Bad}(\bigwedge_{\forall X \in \mathcal{X}^f_\mathcal{Q}, (_,\varphi) \in S^t_X} \neg\varphi)$
$\Leftrightarrow \forall t \in Bad, \mathcal{I} \models \bigwedge_{\forall X \in \mathcal{X}^f_\mathcal{Q}, (_,\varphi) \in S^t_X} \neg\varphi \Leftrightarrow \forall t \in Bad, \mathcal{I} \models \neg \bigvee_{\forall X \in \mathcal{X}^f_\mathcal{Q}, (_,\varphi) \in S^t_X} \varphi$.
Selon la proposition 4.4.3, $(_, \varphi) \in S^t_X \Leftrightarrow t \xrightarrow{\varphi}{}^*_{\mathcal{A}_S} X$,
donc $\forall t \in Bad, \mathcal{I} \models \neg \bigvee_{\forall X \in \mathcal{X}^f_\mathcal{Q}, (_,\varphi) \in S^t_X} \varphi \Leftrightarrow \forall t \in Bad, \mathcal{I} \models \neg \bigvee_{\forall X \in \mathcal{X}^f_\mathcal{Q}, t \xrightarrow{\varphi}{}^*_{\mathcal{A}_S} X} \varphi$.
Et selon la proposition 4.3.5, ceci est équivalent à $t\mathcal{I} \nrightarrow^*_{\mathcal{A}^\mathcal{I}_S} \mathcal{I}(X)$, pour tout $t \in Bad$ et pour tout $X \in \mathcal{X}^f_\mathcal{Q}$. Donc, pour tout X, $\mathcal{I}(X)$ est un état final de $\mathcal{A}^\mathcal{I}_S$. Donc aucun des

termes interdits n'est reconnu par l'instance $\mathcal{A}_S^{\mathcal{I}}$. Nous pouvons alors déduire que ceci est équivalent à l'égalité suivante : $\mathcal{L}(\mathcal{A}_S^{\mathcal{I}}) \cap Bad = \emptyset$. □

4.5.3 Caractérisation d'un point-fixe concluant

Nous sommes désormais proche du but recherché. En effet, pour un STA $\mathcal{A}_S$ donné, un système de réécriture $\mathcal{R}$ et un ensemble de termes interdits Bad, on peut déduire que pour chaque instanciation $\mathcal{I}$ satisfaisant $\phi_{\mathcal{A}_S}^{Bad} \wedge \phi_{\mathcal{R},\mathcal{A}_S}^{FP}$, $\mathcal{R}(\mathcal{L}(\mathcal{A}_S^{\mathcal{I}})) \subseteq \mathcal{L}(\mathcal{A}_S^{\mathcal{I}})$ et $\mathcal{L}(\mathcal{A}_S^{\mathcal{I}}) \cap Bad = \emptyset$. Est-ce suffisant pour garantir que ce point-fixe est intéressant pour les données passées en entrée, *i.e.* $\mathcal{A}$, $\mathcal{R}$ et Bad ? En d'autres mots, peut-on déduire que $\mathcal{R}^*(\mathcal{L}(\mathcal{A})) \cap Bad = \emptyset$ depuis $\mathcal{I} \models \phi_{\mathcal{A}_S}^{Bad} \wedge \phi_{\mathcal{R},\mathcal{A}_S}^{FP}$? Cette réponse est non, étant donné qu'aucune relation n'est spécifiée entre $\mathcal{A}_S$ et $\mathcal{A}$. Nous définissons alors une notion de compatibilité entre $\mathcal{A}_S$ et $\mathcal{A}$, conduisant au résultat que nous attendons.

DÉFINITION 4.5.6 ($\mathcal{A}$-compatibilité)
Soient $\mathcal{A}_S = \langle \mathcal{F}, \mathcal{X}_\mathcal{Q}, \mathcal{X}_\mathcal{Q}^f, \Delta_S \rangle$ un STA, et $\mathcal{A} = \langle \mathcal{F}, \mathcal{Q}, \mathcal{Q}_F, \Delta \rangle$ un automate d'arbres. Le STA $\mathcal{A}_S$ est dit $\mathcal{A}$-compatible si les trois critères suivants sont satisfait :

(1) $\{X_q | q \in \mathcal{Q}\} \subseteq \mathcal{X}_\mathcal{Q}$;

(2) $\{X_q | q \in \mathcal{Q}_f\} \subseteq \mathcal{X}_\mathcal{Q}^f$;

(3) $\{f(X_{q_1}, \ldots, X_{q_n}) \to X_q | f(q_1, \ldots, q_n) \to q \in \Delta\} \subseteq \Delta_S$.

EXEMPLE *4.15*
Soit $\mathcal{A}_1 = \langle \{a, b, s\}, \{q_0, q_1\}, \{q_1\}, \Delta_1 \rangle$ un automate d'arbres tel que :
$\Delta_1 = \{ \quad a \to q_0, b \to q_1,$
$\qquad s(q_0) \to q_1 \ \}.$
Le STA $\mathcal{A}_{S_1}$ de l'exemple 4.3 est alors $\mathcal{A}_1$-compatible. ◀

La notion de $\mathcal{A}$-compatibilité présentée ci-dessus garantit que chaque instance d'un STA $\mathcal{A}_S$ contient le langage $\mathcal{L}(\mathcal{A})$.

PROPOSITION 4.5.7
Soit $\mathcal{A}_S$ un STA et $\mathcal{A}$ un automate d'arbres tel que $\mathcal{A}_S$ est $\mathcal{A}$-compatible. Pour chaque $\mathcal{I} : \mathcal{X}_\mathcal{Q} \mapsto \mathcal{Q}$, on a alors $\mathcal{L}(\mathcal{A}) \subseteq \mathcal{L}(\mathcal{A}_S^{\mathcal{I}})$.

Démonstration. D'après la définition 4.5.6, chaque état q est transformé en un état symbolique X_q. Donc, pour chaque $t \in \mathcal{L}(\mathcal{A}, q_f)$ avec q_f un état final de $\mathcal{A}$, on a alors $t \xrightarrow{\top}{}^*_{\mathcal{A}_S} X_{q_f}$. Donc, pour chaque instanciation $\mathcal{I}$, $\mathcal{I} \models \texttt{Reco}(t, X_{q_f})$. Par conséquent, en appliquant la proposition 4.3.5, on obtient $t \in \mathcal{L}(\mathcal{A}_S^{\mathcal{I}})$. □

Par conséquent, notre principal résultat est la capacité à caractériser par une simple formule de $\rho(\mathcal{X}_\mathcal{Q})$, une approximation point-fixe pouvant être obtenue en utilisant une technique comme la complétion.

THÉORÈME 4.5.8
Soient $\mathcal{A}_S$ un STA et $\mathcal{A}$ un automate d'arbres tels que $\mathcal{A}_S$ est $\mathcal{A}$-compatible. Soient $\mathcal{R}$ un système de réécriture linéaire gauche, Bad un ensemble fini de termes clos, et $\mathcal{I} : \mathcal{X}_\mathcal{Q} \mapsto \mathcal{Q}$ une instanciation. Alors,

$$\mathcal{I} \models \phi_{\mathcal{A}_S}^{Bad} \wedge \phi_{\mathcal{R},\mathcal{A}_S}^{FP} \text{ si et seulement si } \mathcal{A}_S^{\mathcal{I}} \text{ est } \mathcal{R}\text{-clos, } \mathcal{L}(\mathcal{A}) \subseteq \mathcal{L}(\mathcal{A}_S^{\mathcal{I}}) \text{ et } \mathcal{L}(\mathcal{A}_S^{\mathcal{I}}) \cap Bad = \emptyset.$$

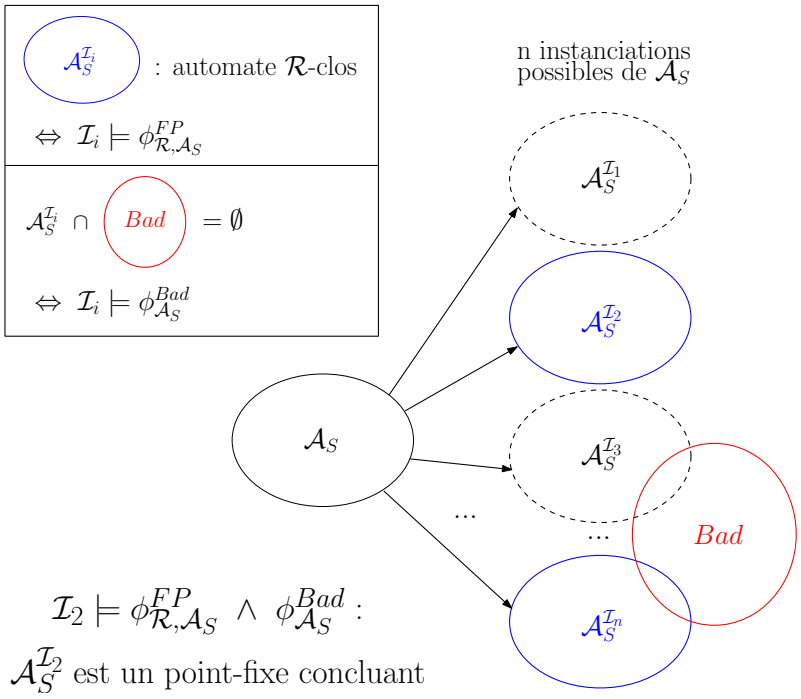


FIGURE 4.8 — Caractérisation d'une approximation point-fixe concluante.

Démonstration. Étant donné que $\mathcal{A}_S$ est $\mathcal{A}-$compatible et $\mathcal{I} \models \phi^{Bad}_{\mathcal{A}_S} \wedge \phi^{FP}_{\mathcal{R},\mathcal{A}_S}$ si et seulement si on a à la fois $\mathcal{I} \models \phi^{Bad}_{\mathcal{A}_S}$ et $\mathcal{I} \models \phi^{FP}_{\mathcal{R},\mathcal{A}_S}$, le théorème 4.5.8 est une conséquence directe des propositions 4.5.2, 4.5.5 et 4.5.7. □

Cette propriété est illustrée sur la figure 4.8, où l'on constate que la seule instance point-fixe concluante apparente est $\mathcal{A}_S^{\mathcal{I}_2}$.

EXEMPLE *4.16*
Dans le cas de l'automate $\mathcal{A}_{S_2}$, la conjonction des deux formules calculées dans les exemples 4.12 et 4.14 est inconsistante. En effet, on a :

$$\begin{aligned}\phi^{FP}_{\mathcal{R},\mathcal{A}_{S_2}} \quad = \quad & \bigwedge_{\forall X \in \mathcal{X}_\mathcal{Q}}((X = X_0) \Rightarrow (((X_0 = X_{nil}) \wedge (X_0 = X)) \\ & \vee((X_p = X_r) \wedge (X_1 = X)))) \\ & \wedge(((X = X_1) \wedge (X_0 = X_{nil})) \Rightarrow (((X_p = X_r) \wedge (X_0 = X_{nil}) \wedge (X_0 = X)) \\ & \vee((X_p = X_r) \wedge (X_1 = X)) \vee ((X_1 = X_{nil}) \wedge (X_0 = X_{nil}) \wedge (X_0 = X)))).\end{aligned}$$

Soit pour simplifier, $\phi^{FP}_{\mathcal{R},\mathcal{A}_{S_2}} = (X_0 = X_1 = X_{nil}) \vee ((X_p = X_r) \wedge ((X_0 = X_{nil}) \vee (X_0 = X_1)))$. Et nous avons également vu que $\phi^{Bad}_{\mathcal{A}_{S_2}} = ((X_0 \neq X_1) \vee (X_0 \neq X_{nil})) \wedge (X_p \neq X_r))$.

Or pour que $\mathcal{A}_{S_2}$ soit un point fixe il doit soit respecter $X_0 = X_1 = X_{nil}$, soit au moins respecter $(X_p = X_r)$, ce qui est incompatible avec $\phi^{Bad}_{\mathcal{A}_{S_2}}$. Il n'y a donc pas de solution et donc aucun point-fixe concluant. ◀

Ce résultat peut également s'interpréter d'une autre manière, formalisée dans le théorème suivant.

THÉORÈME 4.5.9
Soient $\mathcal{A}_S$ un STA et $\mathcal{A}$ un automate d'arbres tels que $\mathcal{A}_S$ est $\mathcal{A}$-compatible. Soient $\mathcal{R}$ un système de réécriture linéaire gauche, Bad un ensemble fini de termes clos, et $\mathcal{I} : \mathcal{X}_\mathcal{Q} \mapsto \mathcal{Q}$

une instanciation. Alors,

$$\mathcal{I} \models \phi^{Bad}_{\mathcal{A}_S} \wedge \phi^{FP}_{\mathcal{R},\mathcal{A}_S} \textit{ implique que } \mathcal{R}^*(\mathcal{L}(\mathcal{A})) \subseteq \mathcal{L}(\mathcal{A}^{\mathcal{I}}_S) \textit{ et } \mathcal{R}^*(\mathcal{L}(\mathcal{A})) \cap Bad = \emptyset.$$

Démonstration. Selon le théorème 4.5.8, $\mathcal{I} \models \phi^{Bad}_{\mathcal{A}_S} \wedge \phi^{FP}_{\mathcal{R},\mathcal{A}_S}$ si et seulement si $\mathcal{A}^{\mathcal{I}}_S$ est $\mathcal{R}-$clos, $\mathcal{L}(\mathcal{A}) \subseteq \mathcal{L}(\mathcal{A}^{\mathcal{I}}_S)$ et $\mathcal{L}(\mathcal{A}^{\mathcal{I}}_S) \cap Bad = \emptyset$. Donc, si $\mathcal{A}^{\mathcal{I}}_S$ est $\mathcal{R}-$ clos et $\mathcal{L}(\mathcal{A}^{\mathcal{I}}_S) \supseteq \mathcal{L}(\mathcal{A})$, alors $\mathcal{L}(\mathcal{A}^{\mathcal{I}}_S) \supseteq \mathcal{R}^*(\mathcal{L}(\mathcal{A}))$ et $\mathcal{R}^*(\mathcal{L}(\mathcal{A})) \cap Bad = \emptyset$. □

Par le biais des formules $\phi^{FP}_{\mathcal{R},\mathcal{A}_S}$ et $\phi^{Bad}_{\mathcal{A}_S}$, la caractérisation de ce qu'est un point-fixe concluant (pour un nombre de transitions donné) est calculée de manière automatique, alors qu'elle est générée manuellement dans les techniques classiques de complétion d'automates d'arbres (par exemple par un ensemble d'équations). Il existe des méthodes de raffinement automatique de l'abstraction si la sur-approximation calculée est trop grossière, mais ces méthodes sont généralement coûteuses (section 3.3.6). Ces formules permettent également d'éviter le raffinement de l'abstraction.

De plus, ces formules constituent une procédure de décision permettant de répondre à la question : "existe-t-il une sur-approximation de l'ensemble des accessibles à au plus n transitions ne contenant aucun terme interdits ?" En effet, ces formules sont calculées pour un STA $\mathcal{A}_S$ donné contenant un ensemble fini n de transitions et compatible avec l'automate de départ. Toutes instanciations possibles de $\mathcal{A}_S$ contiennent donc au plus n transitions et représentent donc l'ensemble des approximations possibles ayant au plus n transitions. Si aucune sur-approximation concluante n'est trouvée parmi toutes ces intanciations, alors on est sur qu'il n'y a aucune sur-approximation de l'ensemble des accessibles à au plus n transitions ne contenant aucun terme interdits.

4.6 Analyse d'atteignabilité via la résolution de formules logiques

Dans cette section, nous synthétisons notre contribution par deux semi-algorithmes. Soient un système de réécriture $\mathcal{R}$, un automate d'arbres $\mathcal{A}$ et un ensemble de termes interdits Bad, le premier semi-algorithme recherche un STA $\mathcal{A}_S$ dont une instance est un point-fixe concluant. Il s'agit donc d'un semi-algorithme étant donné qu'il ne terminera pas si un point-fixe concluant peut ne pas exister (par exemple, si la propriété n'est pas vérifiée), comme démontré dans [Boichut et Héam, 2008]. Dans ce cas, le calcul et la recherche du point-fixe ne terminera pas.

Pour synthétiser son fonctionnement en quelques mots, l'algorithme démarre avec le STA $\mathcal{A}_S$ immédiatement obtenu depuis $\mathcal{A}$ (*i.e.*, $\mathcal{A}$-compatible). Si la formule entière ($\phi^{FP}_{\mathcal{R},\mathcal{A}_S} \wedge \phi^{Bad}_{\mathcal{A}_S}$) ne possède pas de solution (comme dans l'exemple 4.16 de la section précédente), alors le STA courant est amélioré en ajoutant de nouvelles transitions symboliques, en utilisant la fonction `Norm` définie dans l'algorithme. Pour chaque règle de réécriture $l \to r \in \mathcal{R}$, cette fonction va ajouter la normalisation de toutes les parties droites r des règle de réécriture de $\mathcal{R}$ dans l'automate, mais avec **uniquement** des *nouveaux états* (donc sans calcul de substitution ni lien avec ce qui existe déjà dans l'automate). L'intégralité de la formule est calculée pour le nouvel STA et sa satisfiabilité vérifiée grâce à $hasNoValidSolution$. Ce processus est itéré jusqu'à trouver une solution (*i.e.* un point-fixe concluant).

Nous avons utilisé *Mona* [Henriksen et al., 1995] pour résoudre la formule (par le biais de la fonction $hasNoValidSolution$). *Mona* est un outil gérant la logique monadique du second ordre. Pour une formule donnée, *Mona* compile un automate reconnaissant toutes ses solutions. Nous avons alors implémenté un générateur de formules, qui génère automatiquement les formules $\phi^{FP}_{\mathcal{R},\mathcal{A}_S}$ et $\phi^{Bad}_{\mathcal{A}_S}$ dans le format *Mona*, pour un automate d'arbres $\mathcal{A}$, un système de réécriture $\mathcal{R}$ et un ensemble de termes interdits Bad donnés.

Voici maintenant la définition de cet algorithme.

DÉFINITION 4.6.1 (Algorithme de recherche de point-fixe concluant)
Soient un système de réécriture $\mathcal{R}$, un automate d'arbres $\mathcal{A} = \langle \mathcal{F}, \mathcal{Q}, \mathcal{Q}_F, \Delta \rangle$ et un ensemble de termes interdits Bad. Alors `areTermsUnreachable`$?(\mathcal{A}, \mathcal{R}, Bad)$ *est défini comme suit.*

Variables
(STA de départ *)*
$\mathcal{A}_S := \langle \mathcal{F}, \{X_q \mid q \in \mathcal{Q}\}, \{X_q \mid q \in \mathcal{Q}_F\}, \{f(X_{q_1}, \ldots, X_{q_n}) \to X_q | f(q_1, \ldots, q_n) \in \Delta\}\rangle$;
(Formule de départ *)*
$\phi := \phi^{FP}_{\mathcal{R},\mathcal{A}_S} \wedge \phi^{Bad}_{\mathcal{A}_S}$;
00 ***Début***
01 ***Tant que*** *(hasNoValidSolution(ϕ))* ***faire***
02 ***Pour tout*** $l \to r \in \mathcal{R}$ ***faire***
03 $\sigma := \{x_1 \mapsto X_1, \ldots, x_n \mapsto X_n\}$ *avec* $X_1, \ldots, X_n$ *de nouveaux états symboliques*
04 $(\Delta', \mathcal{X}'_\mathcal{Q}) :=$ `Norm`$(r\sigma, X_{n+1})$ *avec* X_{n+1} *un nouvel état symbolique*
05 $\mathcal{A}_S := \langle \mathcal{F}, \mathcal{X}_\mathcal{Q} \cup \mathcal{X}'_\mathcal{Q} \cup \{X_1, \ldots, X_n\}, \mathcal{X}^f_\mathcal{Q}, \Delta' \cup \Delta\rangle$;
06 ***Fin pour tout***
07 $\phi := \phi^{FP}_{\mathcal{R},\mathcal{A}_S} \wedge \phi^{Bad}_{\mathcal{A}_S}$;
08 ***Fin tant que***
09 *retourner vrai;*
10 ***Fin***

La fonction `Norm` *utilisée ligne 04 est définie comme suit :*

$$\texttt{Norm}(t, X) = \begin{cases} (\emptyset, \emptyset), & \textit{si } t \in \mathcal{X}_\mathcal{Q} \\ (\Delta', \mathcal{X}'_\mathcal{Q}) & \textit{si } t = f(t_1, \ldots, t_n) \end{cases}$$

avec $\Delta' = \{f(X_1, \ldots, X_n) \to X\} \cup \bigcup_{i=1}^n (\Delta^i)$,
$\mathcal{X}'_\mathcal{Q} = \{X\} \bigcup_{i=1}^n (\mathcal{X}^i_\mathcal{Q})$,
et `Norm`$(t_i, X_i) = (\Delta^i, \mathcal{X}^i_\mathcal{Q})$.
où X_i est soit un nouvel état symbolique, soit égal à t_i si $t_i \in \mathcal{X}_\mathcal{Q}$.

Nous allons maintenant présenter un exemple complet de recherche d'un point-fixe concluant.

EXEMPLE *4.17*
Soit un système de réécriture $\mathcal{R}_f = \{f(x) \to f(s(s(x)))\}$. Nous voulons ici démontrer que tous les termes de la forme $f(s^{(k)}(0)))$, accessibles depuis le terme $f(0)$ en lui appliquant les règles de $\mathcal{R}$, vérifient la propriété : "k est un nombre pair".

Pour pouvoir faire cette vérification, nous avons alors besoin d'ajouter les trois règles de réécriture suivantes, permettant de modéliser le test de parité :

$$\mathcal{R}_{parity} = \{ \quad even(f(s(s(x)))) \to even(f(x)),$$
$$even(f(0)) \to true,$$
$$even(f(s(0))) \to false\}.$$

Les données passées en entrée de l'algorithme 4.6.1 sont donc : le système de réécriture $\mathcal{R} = \mathcal{R}_f \cup \mathcal{R}_{parity}$, $Bad = \{false\}$ et $\mathcal{A} = \langle \mathcal{F}, \mathcal{Q}, \mathcal{Q}^f, \Delta \rangle$ avec $\mathcal{Q} = \{q_0, q_1, q_2\}$, $\mathcal{F} = \{f : 1, s : 1, 0 : 0, even : 1, true : 0, false : 0\}$, $\mathcal{Q}^f = \{q_2\}$ et :
$\Delta = \{even(q_1) \to q_2, f(q_0) \to q_1, 0 \to q_0\}$.

Si une sur-approximation concluante (*i.e.* un automate point-fixe qui n'accepte aucun terme interdit) peut être trouvée par cet algorithme, alors cela garantit que la propriété à vérifier est vraie, *i.e.* que l'ensemble des termes accessibles depuis le terme $f(0)$ en utilisant $f(x) \to f(s(s(x)))$ sont nécessairement de la forme $f(s^k(0))$ avec k un entier pair.

Le STA de départ, étant $\mathcal{A}$-compatible, comprend alors l'ensemble de transitions symboliques suivant :
$\Delta = \{even(X_1) \to X_2, 0 \to X_0, f(X_0) \to X_1\}$,
avec $\mathcal{X}_\mathcal{Q} = \{X_0, X_1, X_2\}$. En appliquant la définition 4.5.1, on obtient alors la formule suivante :

$\phi^{FP}_{\mathcal{R},\mathcal{A}_S} = \bigwedge_{Y \in \{X_1,X_2,X_3\}} (\top \wedge X_1 = Y \Rightarrow \bot) \wedge$ $\bigwedge_{X,Y \in \{X_1,X_2,X_3\}} (\bot \Rightarrow \top \wedge X = Y) \wedge$ $\bigwedge_{Y \in \{X_1,X_2,X_3\}} (\top \wedge X_2 = Y \Rightarrow \bot) \wedge$ $\bot \Rightarrow \bot$	$\mathcal{R} =$ $f(x) \to f(s(s(x)))$ $even(f(s(s(x)))) \to even(f(x))$ $even(f(0)) \to true$ $even(f(s(0))) \to false$

On constate ici que $\phi^{FP}_{\mathcal{R},\mathcal{A}_S}$ n'est pas satisfiable. En effet, $\phi^{FP}_{\mathcal{R},\mathcal{A}_S} = \phi_1 \wedge (\top \wedge X_1 = X_1 \Rightarrow \bot) \wedge \phi_2$ avec $\phi_1, \phi_2 \in \rho(\mathcal{X}_\mathcal{Q})$. En simplifiant la formule, on obtient que $\phi^{FP}_{\mathcal{R},\mathcal{A}_S} = \phi_1 \wedge (\top \Rightarrow \bot) \wedge \phi_2 = \bot$. Donc ϕ (voir ligne 01 de l'algorithme 4.6.1) est également insatisfiable. Par conséquent, il n'est donc pas nécessaire de calculer $\phi^{Bad}_{\mathcal{A}'_S}$ pour savoir que le STA $\mathcal{A}_S$ nécessite d'être étendu. Dans cet exemple, quatre substitutions (une par règle de réécriture, en suivant l'ordre du tableau ci-dessus) σ_1 à σ_4 sont créées telles que $\sigma_1 = \{x \mapsto X_4\}$, $\sigma_2 = \{x \mapsto X_8\}$ et $\sigma_3 = \sigma_4 = \emptyset$ avec X_4 et X_8 deux *nouveaux* états symboliques.

Par conséquent, en appliquant ces quatre substitutions dans l'ordre, respectivement sur les quatre termes $f(s(s(x)))$, $even(f(x))$, $true$ et $false$, on doit alors ajouter les nouveaux termes $f(s(s(X_4)))$, $even(f(X_8))$, $true$ et $false$ dans l'automate, en les normalisant. Soient X_3, X_7, X_{10} et X_{11} quatre nouveaux états symboliques, l'étape de normalisation de la ligne 04 – $\texttt{Norm}(f(s(s(X_4))), X_3)$, $\texttt{Norm}(even(\ f(X_8)), X_7)$, $\texttt{Norm}(true, X_{10})$ et $\texttt{Norm}(false, X_{11})$ – peut alors produire le STA suivant :
$\mathcal{A}'_S = \langle \mathcal{F}, \mathcal{X}_\mathcal{Q}, \mathcal{X}^f_\mathcal{Q}, \Delta \rangle$ avec $\mathcal{X}_\mathcal{Q} = \{X_0, \ldots, X_{11}\}$, $\mathcal{X}^f_\mathcal{Q} = \{X_2\}$ et $\Delta = \{true \to X_{10}, false \to X_{11}, s(X_5) \to X_6, s(X_4) \to X_5, 0 \to X_0, even(X_9) \to X_7, even(X_1) \to X_2, f(X_8) \to X_9, f(X_6) \to X_3, f(X_0) \to X_1\}$. Nous pouvons ici noter que $\mathcal{A}'_S$ est également $\mathcal{A}$-compatible.

Nous devons ensuite calculer la formule permettant de trouver une instance du STA ne comportant aucun terme interdit. Rappelons ici que $Bad = \{false\}$. En suivant la définition 4.5.4, on obtient alors la formule suivante :

$$\phi^{Bad}_{\mathcal{A}'_S} = X_2 \neq X_{11}.$$

```
MONA v1.4-13 for WS1S/WS2S
Copyright (C) 1997-2008 BRICS

PARSING
Time: 00:00:00.00

CODE GENERATION
DAG hits: 5549, nodes: 447
Time: 00:00:00.01
AUTOMATON CONSTRUCTION
100% completed
Time: 00:00:00.22

Automaton has 764 states and 27075
BDD-nodes

REDUCTION
Projections removed: 2 (of 3)
Products removed: 60 (of 433)
Other nodes removed: 1 (of 7)
DAG nodes after reduction: 380
Time: 00:00:00.00

...
Xq11 = {1}, Xq10 = {0}, Xq9 = {1}, Xq8 = {1}
Xq7 = {1}, Xq6 = {1}, Xq5 = {0}, Xq4 = {1}
Xq3 = {1}, Xq2 = {0}, Xq1 = {0}, Xq0 = {0}

Total time: 00:00:00.25
```

FIGURE 4.9 — Sortie du programme *Mona* après exécution.

Pour trouver une solution, *i.e.* un point fixe concluant, nous devons également résoudre la formule $\phi^{FP}_{\mathcal{R},\mathcal{A}_S}$. Cette formule, ainsi que le programme *Mona* correspondant à notre exemple, peuvent être téléchargés à l'adresse suivante : http://www.univ-orleans.fr/lifo/Members/Yohan.Boichut/research/exampleMona.txt.

Cette formule est composée de 93 implications, *i.e.* de la forme $((A_1 \wedge \ldots \wedge A_n) \Rightarrow (B_1 \wedge \ldots \wedge B_m))$. Nous constatons que pour un STA $\mathcal{A}_S$ donné et un système de réécriture $\mathcal{R}$, la taille de cette formule peut être énorme et la résolution de telles formules nécessite des techniques de résolutions dédiées. La figure 4.9 présente la sortie résultant de l'exécution de ce programme *Mona*.

Construisons l'instanciation $\mathcal{I}$ provenant de la solution retournée par l'exécution de notre programme *Mona*. On obtient alors :

$$\begin{aligned}\mathcal{I} = \quad & \{X_0 \mapsto q_0, X_1 \mapsto q_0, X_2 \mapsto q_0, X_3 \mapsto q_1,\\ & X_4 \mapsto q_1, X_5 \mapsto q_0, X_6 \mapsto q_1, X_7 \mapsto q_1,\\ & X_8 \mapsto q_1, X_9 \mapsto q_1, X_{10} \mapsto q_0, X_{11} \mapsto q_1\}.\end{aligned}$$

Si l'on applique $\mathcal{I}$ sur le STA $\mathcal{A}_S$, on obtient alors l'automate d'arbres $\mathcal{A}^{\mathcal{I}}_S$ suivant :
$\mathcal{A}^{\mathcal{I}}_S = \langle \mathcal{F}, \{q_0, q_1\}, \{q_0\}, \Delta^{\mathcal{I}} \rangle$ avec

$$\begin{aligned}\Delta^{\mathcal{I}} = \{ \quad & false \to q_1, 0 \to q_0,\\ & s(q_0) \to q_1, s(q_1) \to q_0,\\ & even(q_1) \to q_1, even(q_0) \to q_0,\\ & true \to q_0, f(q_1) \to q_1, f(q_0) \to q_0\}.\end{aligned}$$

On peut constater que cet automate d'arbres est $\mathcal{R}$-clos. En effet, pour la règle $f(x) \to f(s(s(x)))$, on peut observer que $f(q_0)$ et $f(s(s(q_0)))$ peuvent être réduit en q_0, donc $f(s(s(q_0)))$ appartient déjà au langage de l'automate. De façon similaire, $f(q_1)$ et $f(s(s(q_1)))$ peuvent réduit sur q_1. De la même manière, pour la règle $even(f(s(s(x)))) \to even(f(x))$, on remarque que $even(f(s(s(q_0)))) \to^*_{\mathcal{A}^{\mathcal{I}}_S} q_0$ et $even(q_0) \to^*_{\mathcal{A}_S} q_0$. Et on a également $even(f(s(s(q_1)))) \to^*_{\mathcal{A}^{\mathcal{I}}_S} q_1$ et $even(q_1) \to^*_{\mathcal{A}^{\mathcal{I}}_S} q_1$. Finalement, pour la règle $even(f(0)) \to true$, on a $even(f(0)) \to^*_{\mathcal{A}^{\mathcal{I}}_S} q_0$ et $true \to^*_{\mathcal{A}^{\mathcal{I}}_S} q_0$.

De plus, le terme $false$ n'appartient pas à $\mathcal{L}(\mathcal{A}^{\mathcal{I}}_S)$. L'automate $\mathcal{A}^{\mathcal{I}}_s$ est donc un point-fixe concluant.

◀

Des quelques expérimentations que nous avons menées grâce à l'outil *Mona* res-

sortent les points suivants :

- *Mona* ne peut pas gérer les formules générées par des spécifications comprenant plus de 20 variables (ou états symboliques) différentes.
- La taille des fichiers *Mona* contenant les formules générés grâce à notre implémentation dépasse parfois plusieurs méga-octets. La taille du fichier généré par l'exemple précédent est de 20 Ko.

Une propriété intéressante à obtenir pour cet algorithme est la certitude, pour un système donné, d'obtenir une sur-approximation régulière n'admettant aucun terme interdit (*i.e.* concluante), si cette dernière existe pour le système vérifié.

Pour le moment, nous n'avons pas encore obtenu de preuve formelle de cette propriété pour l'algorithme 4.6.1. Cependant, nous avons obtenu cette preuve pour un algorithme équivalent, mais cet algorithme génère des formules plus importantes que l'algorithme 4.6.1. Le nombre de variables est également considérablement augmenté. Soit $\mathcal{F}$ l'alphabet fonctionnel du système. Tant qu'aucune solution point-fixe concluante n'est trouvée, ce deuxième algorithme ajoute, à chaque étape, une nouvelle transition par symbole de $\mathcal{F}$, avec de nouveaux états symboliques (*i.e.* inexistants dans l'automate précédant).

EXEMPLE *4.18*
Soient $\mathcal{A}_0 = \{\mathcal{F}, \mathcal{XQ}_0, \mathcal{X}_{Q_{f_0}}, \Delta_0\}$ le STA de départ avec $\mathcal{F} = \{a_0, f_2\}$, et $X'_0, X'_1, X'_2, X' \notin \mathcal{X}_\mathcal{Q}$ de nouveaux états symboliques. Alors dans la première étape de l'algorithme, les transitions $a \to X'_0$ et $f(X'_1, X'_2) \to X'$ seront ajoutées. ◀

Il s'agit donc également d'un semi-algorithme, étant donné qu'il ne termine pas si aucune solution n'existe.

DÉFINITION 4.6.2 (Deuxième algorithme de recherche de point-fixe concluant)
Soient un système de réécriture $\mathcal{R}$, un automate d'arbres $\mathcal{A} = \langle \mathcal{F}, \mathcal{Q}, \mathcal{Q}_F, \Delta \rangle$ et un ensemble de termes interdits Bad. Alors `areTermsUnreachable?2`$(\mathcal{A}, \mathcal{R}, Bad)$ *est défini comme suit.*

Variables
(STA et formule de départ *)*
$\mathcal{A}_S := \langle \mathcal{F}, \mathcal{X}_\mathcal{Q} = \{X_q \mid q \in \mathcal{Q}\}, \{X_q \mid q \in \mathcal{Q}_F\}, \Delta_S = \{f(X_{q_1}, \ldots, X_{q_n}) \to X_q | f(q_1, \ldots, q_n) \in \Delta\}\rangle$;
$\phi := \phi^{FP}_{\mathcal{R},\mathcal{A}_S} \wedge \phi^{Bad}_{\mathcal{A}_S}$;
(Arité maximum de l'alphabet $\mathcal{F}$ *)*
$n := arite_max(\mathcal{F})$;
00 ***Début***
01 ***Tant que*** *(hasNoValidSolution(ϕ))* ***faire***
02 ***Pour*** i *de 0 à* n ***faire***
03 ***Pour tout*** $f \in \mathcal{F}^i$ ***faire***
04 $\Delta_S := \Delta_S \cup \{f(X'_1, \ldots, X'_i) \to X'\}$,
avec $X'_1, \ldots, X'_i, X' \notin \mathcal{X}_\mathcal{Q}$ *de nouveaux états symboliques;*
05 $\mathcal{X}_\mathcal{Q} = \mathcal{X}_\mathcal{Q} \cup \{X'_1, \ldots, X'_i, X'\}$
06 ***Fin pour tout***
07 ***Fin pour***
08 $\phi := \phi^{FP}_{\mathcal{R},\mathcal{A}_S} \wedge \phi^{Bad}_{\mathcal{A}_S}$;
09 ***Fin tant que***
10 *retourner vrai;*
11 ***Fin***

Si le système admet une approximation régulière qui vérifie la propriété, alors l'algorithme `areTermsUnreachable?2`$(\mathcal{A}, \mathcal{R}, Bad)$ de la définition 4.6.2 termine et trouve nécessairement une approximation point-fixe concluante. Nous allons formaliser cette propriété dans le théorème suivant.

THÉORÈME 4.6.3
Soient $\mathcal{A}$ un automate d'arbres, $\mathcal{R}$ un système de réécriture et Bad un ensemble fini de termes interdits.

S'il existe au moins un automate d'arbres $\mathcal{A}'$ tel que $\mathcal{A}' \supseteq \mathcal{R}^*(\mathcal{L}(\mathcal{A}))$ et $\mathcal{A}' \cap Bad = \emptyset$, alors `areTermsUnreachable?2`$(\mathcal{A}, \mathcal{R}, Bad)$ *est terminant et l'automate d'arbres $\mathcal{A}''$ retourné par cet algorithme vérifie : $\mathcal{A}'' \supseteq \mathcal{R}^*(\mathcal{L}(\mathcal{A}))$ et $\mathcal{A}'' \cap Bad = \emptyset$.*

Démonstration. Soient $\mathcal{A} = \{\mathcal{F}, \mathcal{Q}, \mathcal{Q}_f, \Delta\}$ et $\mathcal{A}' = \{\mathcal{F}, \mathcal{Q}', \mathcal{Q}'_f, \Delta'\}$ des automates d'arbres et $\mathcal{R}$ un système de réécriture tels que $\mathcal{A}' \supseteq \mathcal{R}^*(\mathcal{L}(\mathcal{A}))$ et $\mathcal{A}' \cap Bad = \emptyset$ (*i.e.*, $\mathcal{A}'$ est un point-fixe concluant).

Soit $\mathcal{F} = \{f_1, \ldots, f_n\}$. Alors Δ' est nécessairement de la forme :

$$\begin{array}{llll}
\{ & f_1(q_{1,1}, \ldots, q_{1,i_1}) \to q_1, & \ldots, & f_1(q_{1,j_1}, \ldots, q_{1,k_1}) \to q_{nb_1}, \\
 & f_2(q_{2,1}, \ldots, q_{2,i_2}) \to q_{nb_1+1}, & \ldots, & f_2(q_{2,j_2}, \ldots, q_{2,k_2}) \to q_{nb_1+nb_2}, \\
 & \ldots, & & \\
 & f_n(q_{n,1}, \ldots, q_{1,i_n}) \to q_{nb_1+\ldots+nb_{n-1}+1}, & \ldots, & f_n(q_{n,j_n}, \ldots, q_{1,k_n}) \to q_{nb_1+\ldots+nb_n} \quad \},
\end{array}$$

avec, pour $p \in [1, n]$, i_p l'arité du symbole f_p (*i.e.* $k_p - j_p = i_p$), et nb_p le nombre de transitions concernant le symbole f_p. Autrement dit, les transitions de $\mathcal{A}'$ ne sont composées que de symboles de $\mathcal{F}$, et pour chaque symbole de $\mathcal{F}$, il y a nécessairement 0 ou plusieurs transitions le concernant.

Soit $f_{max} \in \mathcal{F}$ le symbole dont les transitions apparaissent le plus dans Δ'. Il y a donc nb_{max} transitions commençant par f_{max} dans $\mathcal{A}'$.

L'algorithme `areTermsUnreachable?2`$(\mathcal{A}, \mathcal{R}, Bad)$ consiste à ajouter à chaque tour de boucle, une transition de chaque symbole de $\mathcal{F}$, jusqu'à ce qu'il trouve un point-fixe concluant (ou bien il tourne à l'infini s'il n'en trouve pas). Au bout de nb_{max} tours de boucle, on obtient alors un STA $\mathcal{A}'_S$ qui possède alors au moins nb_{max} transitions pour chaque symbole de $\mathcal{F}$, soit le nombre de transitions nécessaires pour pouvoir obtenir $\mathcal{A}'$, puisque le nombre maximum de transitions de $\mathcal{A}'$ par symbole est nb_{max}.

Les nb_{max} transitions par symbole ajoutées sont toutes crées avec de nouveaux états symboliques, *i.e.* sans aucune contrainte d'instanciation. Ces états symboliques peuvent dont être instanciés par n'importe quels états, et peuvent donc être instanciés par les états de $\mathcal{A}'$ afin que les transitions de $\mathcal{A}'_S$ correspondent aux transitions de $\mathcal{A}'$. Les transitions inutiles de $\mathcal{A}'_S$ pointent alors sur des états instanciés pour être non-joignables. Ceci est possible car $\mathcal{A}'_S$ est calculé au bout de nb_{max} tours, et possède donc forcément le nombre de transitions nécessaires.

Au moment de la recherche d'une instance solution (ligne 01), au bout de nb_{max} tours, l'algorithme `areTermsUnreachable?2`$(\mathcal{A}, \mathcal{R}, Bad)$ va donc nécessairement trouver l'automate $\mathcal{A}'$ ou un autre automate $\mathcal{A}''$ point-fixe concluant.

□

4.7 Conclusion et perspectives

Ce chapitre définit tout d'abord un nouveau type d'automate d'arbres appelés automates d'arbres symboliques (ou STA), dont les états sont des variables instanciables (ou états symboliques). Nous avons ensuite écrit un nouvel algorithme de calcul de substitutions adapté à ces nouveaux automates. Ces substitutions sont accompagnées de formules logiques décrivant sous quelles conditions ces substitutions peuvent être effectuées. Les formules logiques utilisées sont des conjonctions ou disjonctions d'égalités entre les états symboliques du STA.

Pour un STA $\mathcal{A}_S$ donné, un ensemble de termes interdits Bad, un automate d'arbres $\mathcal{A}$ et un système de réécriture $\mathcal{R}$, nous avons ensuite caractérisé, grâce à deux formules logiques calculées *automatiquement* à partir des données passées en entrée et utilisant l'algorithme de filtrage précédemment défini, ce qu'était un *point-fixe concluant* dans le cadre de l'analyse d'atteignabilité. Chaque solution respectant ces deux formules est une instanciation pouvant être appliquée au STA $\mathcal{A}_S$, produisant alors la sur-approximation souhaitée. Nous avons finalement défini différents semi-algorithmes permettant d'itérer le processus de recherche de point-fixe, en enrichissant le langage du STA à chaque étape de l'algorithme, permettant ainsi de se rapprocher d'une solution. La résolution des formules, ainsi qu'un des algorithmes de recherche ont été implémentés en utilisant l'outil *Mona*.

L'automate d'arbres obtenu de manière automatique par notre méthode, aurait pu être calculé par la technique de complétion classique que l'on trouve par exemple dans [Feuillade et al., 2004]. Mais une telle technique nécessite un paramètre technique (i.e, une fonction d'approximation), qui influe grandement sur la qualité de l'approximation calculée. Ce paramètre doit être défini manuellement et nécessite donc une certaine expertise.

Dans notre cas, un STA généré depuis un automate d'arbres $\mathcal{A}$ initial permet de calculer automatiquement les formules logiques et de rechercher une solution. Si aucune solution n'est trouvée à une étape donnée, alors nous sommes certains qu'il n'existe aucun automate $\mathcal{R}$-clos et concluant à cette étape du calcul. La taille du STA courant est alors augmentée jusqu'à obtention d'une solution. Aucun paramétrage n'est alors nécessaire, et la recherche d'une solution garantit l'absence de faux contre-exemples, contrairement aux techniques d'abstractions usuelles.

De plus, nous avons vu en fin de section 4.5 que les formules logiques calculées constituent une procédure de décision permettant de répondre à la question : "existe-t-il une sur-approximation de l'ensemble des accessibles à au plus n états ne contenant aucun terme interdits ?". Ceci est renforcé par le théorème 4.6.3, qui démontre que, *si* une sur-approximation régulière de l'ensemble des termes accessibles qui n'accepte aucun terme interdit *existe pour le système vérifié*, alors l'algorithme 4.6.2 permet théoriquement de la trouver.

Ce travail est une première étape vers une technique de vérification basée sur la résolution de formules logiques. En effet, les spécifications contenant plus de 20 états symboliques sont hors de la portée de l'outil *Mona*. Même si les formules engendrées par de telles spécifications présentent une certaine régularité dans leur forme, leur taille peut être énorme (en particulier pour la formule $\phi^{FP}_{\mathcal{R},\mathcal{A}_S}$ de la définition 4.5.1).

La méthode définie dans ce chapitre s'applique à un ensemble *fini* de termes interdits Bad (voir sous-section 4.5.2). En effet, la formule $\phi^{Bad}_{\mathcal{A}_S}$ est générée en calculant, pour

chaque $t \in Bad$, sa condition de non-réduction dans $\mathcal{A}_S$. Si Bad était un ensemble de termes *infini*, le calcul de $\phi^{Bad}_{\mathcal{A}_S}$ ne pourrait pas converger. Une perspective serait donc d'étendre cette méthode à un ensemble infini de termes interdits, représenté par un automate d'arbres $\mathcal{A}_{Bad}$. Il faudrait alors générer une formule signifiant que l'intersection $\mathcal{A}_S \cap \mathcal{A}_{Bad}$ est vide. On pourrait pour cela utiliser le travail effectué dans [Boichut et al., 2012], où l'intersection de deux automates est modélisée par une formule logique.

Pour résoudre le problème de la grande taille des formules générées, le travail en cours est basé sur des techniques de résolutions de contraintes dédiées et sur la recherche d'heuristiques qui nous permettraient de manipuler des formules de taille importante. La formule la plus importante en terme de taille est la formule point-fixe $\phi^{FP}_{\mathcal{R},\mathcal{A}_S}$. Une idée serait alors, en se basant la formule $\phi^{Bad}_{\mathcal{A}_S}$ générée au préalable, de construire la formule $\phi^{FP}_{\mathcal{R},\mathcal{A}_S}$ d'une manière plus intelligente. Cette formule se construit principalement grâce à l'algorithme de filtrage : on pourrait alors, lors de la construction des formules durant l'algorithme, couper des branches inconsistantes à la volée en propageant les contraintes des termes interdits. Pour cela, une technique *à la Mona* est en cours d'étude.

Malgré que la grande taille des formules générées soit un obstacle quant à l'efficacité de cette technique, elle permet de donner une indication concrète de la complexité du problème d'atteignabilité dans le cas de systèmes à états infinis. Si l'on attaque ce problème d'atteignabilité de front sans aucun paramètre (*i.e.* aucune fonction d'abstraction de départ entrée par un utilisateur expérimenté), on constate que ce problème est difficile à résoudre puisque générant des formules de très grande taille même sur de petits exemples. En effet, dans la section 4.6, la formule générée comporte 93 implications pour un exemple gadget.

Ce travail permet donc réellement de quantifier la difficulté du problème auquel on s'attaque dans cette thèse, et il permet également de démontrer que la technique actuelle, se basant sur des fonctions d'abstractions données au départ telles qu'un ensemble d'équations, est une bonne solution. C'est donc cette solution que nous allons choisir d'améliorer dans la deuxième partie de cette thèse.

Deuxième partie

Automates d'arbres à treillis et interprétation abstraite

Algorithme de complétion pour automates d'arbres à treillis

5

Sommaire

5.1 Introduction **143**

5.2 Automates d'arbres à treillis (Lattice Tree Automata : *LTA*) **147**

5.2.1 Symboles interprétables et évaluation 147

5.2.2 Définition et sémantique 149

5.3 Opérations sur les LTA **152**

5.3.1 Opérations booléennes 152

5.3.2 Déterminisation 154

5.3.3 Raffinement de la partition 159

5.4 Un algorithme de complétion pour les LTA **160**

5.4.1 Calcul du LTA successeur 162

5.4.2 Évaluation d'un LTA 166

5.4.3 Abstraction équationnelle 167

5.4.4 Algorithme complet de complétion pour LTA et preuve de correction 170

5.5 Complétion pour LTA déroulée sur un exemple **174**

5.6 Conclusion **177**

"Le monde doit tous ses progrès aux gens mal à l'aise.
L'homme heureux ne se hasarde pas à dépasser les limites anciennes."

Nathaniel Hawthorne

Résumé

Nous allons voir dans ce chapitre que les techniques actuelles liées au Model-Checking régulier sur arbres ne permettent pas de capturer efficacement à la fois la structure complexe d'un système et certaines de ses caractéristiques. Si on prend par exemple les programmes $Java$, la structure d'un terme est efficacement exploitée pour modéliser la structure d'un état du système. En contrepartie, les entiers présents dans les programmes $Java$ doivent être encodés par des entiers de Peano, et dans ce cas chaque opération arithmétique est potentiellement modélisée par une centaine d'applications de règles de réécriture. Dans ce chapitre, nous proposons les *automates d'arbres à treillis* (LTA), une version étendue des automates d'arbres dont les feuilles sont équipées avec des éléments d'un treillis.

Les LTA nous permettent de représenter des ensembles possiblement infinis de termes pouvant être interprétés. Ces termes *interprétables* permettent de représenter efficacement des domaines complexes et leurs opérations associées. Nous étendons également les opérations booléennes des automates d'arbres classiques aux LTA. Enfin, en tant que contribution principale, nous définissons un nouvel algorithme de complétion, équipé d'une fonction d'abstraction, permettant de calculer l'ensemble possiblement infini des termes interprétables accessibles en un temps fini.

5.1 Introduction

Nous avons vu, dans les chapitres précédents de ce mémoire, que la modélisation grâce à un espace infini d'états était utilisée pour éviter les hypothèses artificielles sur les structures de données, comme par exemple décider d'une borne maximum arbitraire sur la taille des piles ou la valeur des entiers. Nous utilisons alors dans cette thèse une représentation symbolique pour modéliser des espaces infinis d'états : les automates d'arbres (voir définition 2.2.3).

Rappelons brièvement que cette représentation est utilisée dans le but de vérifier des propriétés de sûreté sur les systèmes représentés, et que la technique de vérification utilisée est le Model-Checking régulier d'arbres (Regular Tree Model-Checking : RTMC). Dans cette technique, les états du système sont représentés par des automates d'arbres, et son comportement par un système de réécriture. Des techniques d'abstraction et d'accélération du calcul des états accessibles sont très souvent utilisées. Parmi ces techniques d'abstraction se trouve la fusion d'états équivalents par équations (l'*abstraction équationnelle* : section 2.4.2), dont il sera question dans ce chapitre.

Dans [Boichut et al., 2007], les auteurs proposent une traduction *exacte* de la sémantique de la machine $Java$ virtuelle en utilisant des automate d'arbres et des règles de réécriture. Cette traduction permet d'analyser des programmes $Java$ grâce aux model-checkers RTMC classiques. Une des principales difficultés de ce codage est de capturer et gérer les deux dimensions infinies qui peuvent apparaitre dans un programme $Java$.

En effet, le comportement infini de ces programmes peut être dû à un nombre non-borné d'appels de méthodes ou de création d'objets, ou simplement parce que le programme manipule des données dont le domaine n'est pas borné, comme par exemple les variables entières. Or, dans le cadre du RTMC, la représentation actuelle de ces données définies sur un domaine infini alourdit significativement le temps de calcul, alors que de multiples comportements infinis sur la structure du programme peuvent être sur-approchés de manière très efficace par la complétion d'automates d'arbres et l'abstraction équationnelle [Genet et Rusu, 2010].

Par exemple, dans [Boichut et al., 2007], la représentation par termes permet de représenter la structure d'une configuration d'une manière très concise. En effet, les piles peuvent se modéliser facilement par des termes (on peut par exemple avoir la pile `stack(1,stack(2,stack(3,nil))))`, ou encore, une méthode se représente facilement par le terme `frame(m,pc,s,l)` où `m` est le nom de la méthode, `pc` le point de programme courant, `s` la pile des instructions, et `t` le tableau des variables locales.

Mais d'un autre côté, la représentation des entiers relatifs et leurs différentes opérations arithmétiques n'est pas optimale et nécessite d'être simplifiée. En effet, pour pouvoir modéliser un entier par un terme, et une opération arithmétique par un ensemble de règles de réécriture, on peut par exemple utiliser l'arithmétique de Peano, où les entiers sont représentés grâce à trois fonctions :

- `succ(n)`, représentant l'entier successeur de `n`,
- `pred(n)` l'entier prédécesseur de `n`, et
- `zero` l'entier 0.

Par exemple, `succ(succ(zero))` représente l'entier 2, et `pred(zero)` l'entier -1. Les opérations arithmétiques peuvent ainsi se faire grâce à des systèmes de réécriture. Les règles de réécriture représentant l'addition de deux entiers sont représentées en figure 5.1.

```
xadd(zero,zero)          →  result(zero)
xadd(succ(a),pred(b))    →  xadd(a,b)
xadd(pred(a),succ(b))    →  xadd(a,b)
xadd(succ(a),succ(b))    →  xadd(succ(succ(a)),b)
xadd(pred(a),pred(b))    →  xadd(pred(pred(a)),b)
xadd(succ(a),zero)       →  result(succ(a))
xadd(pred(a),zero)       →  result(pred(a))
xadd(zero,succ(b))       →  result(succ(b))
xadd(zero,pred(b))       →  result(pred(b))
```

$$\texttt{xadd(succ(zero), succ(zero))} \xrightarrow{1} \texttt{xadd(succ(succ(zero)),zero)} \xrightarrow{2} \texttt{result(succ(succ(zero)))}$$

FIGURE 5.1 — Règles de réécriture représentant l'addition de deux entiers relatifs, et exemple de l'addition $1 + 1$.

Mais cette représentation a un impact exponentiel sur la taille des automates représentant les ensembles d'états du programme aussi bien que sur le temps de calcul des états accessibles. Par exemple, le temps de calcul de "$300 + 400$", représenté par le terme $xadd(succ^{300}(zero), succ^{400}(zero))$, nécessite d'appliquer 300 fois les règles de réécritures de la figure 5.1 ($x + y$ requiert l'application de x règles de réécriture), et donc d'autant d'étapes de complétion. De la même manière, $x \times y$ requiert l'application de $x \times (y + 2)$ règles de réécriture.

On peut également représenter les entiers par des termes en utilisant une liste de bits, et dans ce cas, les opérations arithmétiques sont modélisées par des règles de réécriture codant les opérations binaires. Par exemple, l'entier 5 est représenté par le terme $stack(un, stack(un, stack(zero, nil)))$ (représentant le nombre binaire 110). Cette représentation, bien que plus concise que la précédente, surcharge également la taille des automates et du système de réécriture, ainsi que le calcul des accessibles. Ainsi, une addition $x + y$ implique l'application de $log(x)$ règles de réécriture, ce qui est plus efficace, mais grandement perfectible.

Pour résoudre ce problème, et minimiser le nombre d'étapes de réécriture, on peut suivre la solution trouvée par [Kaplan et Choppy, 1989]. Dans ces travaux, l'idée est de pouvoir filtrer un terme contenant directement des entiers tel que $add(2, 3)$ pour l'application des règles d'addition telles qu'indiquées en figure 5.1, sans avoir besoin d'indiquer le terme $add(s(s(zero)), s(s(s(zero))))$. Le filtrage doit ainsi être associé à une opération d'évaluation permettant d'appliquer pour le terme $add(2, 3)$ la règle de réécriture $add(x, s(y)) \rightarrow add(s(x), y)$. Ceci est effectué grâce à une fonction inverse permettant de détecter que $3 = s(2)$, $2 = s(1)$, etc. Cette fonction inverse est fournie par une spécification *built-in*, donnée par l'utilisateur, qui peut également définir des opérations built-in telles que les opérations arithmétiques afin de pouvoir les évaluer directement, sans passer l'application de plusieurs règles de réécriture comme c'est le cas en figure 5.1.

Dans ce cas précis, le terme possédant une opération *built-in* doit alors être *interprété* (ou *évalué*) et retourner directement le résultat de l'opération décrite, sans appliquer aucune règle de réécriture. Ainsi $2+3$ devient un terme qui est directement évalué en 5. Ainsi, le but de ce chapitre est l'étude de nouvelles approches de Model-Checking pour

de tels termes *interprétables*. De plus, contrairement à [Kaplan et Choppy, 1989], le but est également de pouvoir filtrer des termes comprenant des éléments d'un domaine infini tel que $\mathbb{Z}$, mais sans avoir recours à une quelconque fonction inverse, grâce à l'intégration dans le système de réécriture, ainsi que dans les automates d'arbres, les éléments d'un tel domaine infini.

La première contribution de ce chapitre est la définition des *automates d'arbres à treillis* (*Lattice Tree Automata* : LTA), une nouvelle classe d'automates d'arbres capable de représenter des ensembles possiblement infinis de termes *interprétables*. Intuitivement, les LTA sont des automates d'arbres dont les feuilles peuvent être équipées d'un élément d'un treillis Λ, pouvant abstraire un ensemble possiblement infini de valeurs d'un domaine concret $\mathcal{C}$. Les éléments du treillis sont alors des constantes de l'alphabet (ici *infini*) utilisé par le langage de l'automate. Par exemple, on peut abstraire un ensemble de valeurs entières (*i.e.* $\mathcal{C} = \mathbb{Z}$) par un intervalle au lieu d'utiliser un codage unaire ou binaire de ces entiers comme dans [Boichut et al., 2007]. Les noeuds des LTA peuvent être soit des symboles non-interprétables (*i.e.* des symboles classiques), soit représenter des opérations arithmétiques. Ainsi, l'algorithme de complétion associé permettra de calculer chaque opération arithmétique en *une seule* étape de complétion (contrairement au cas Peano décrit ci-dessus), grâce à des techniques issues de l'*interprétation abstraite* [Cousot et Cousot, 1977] (voir section 2.6).

Nous proposons ensuite dans la section 5.3, l'étude et la définition, pour les LTA, de toutes les opérations classiques des automates (intersection, déterminisation, etc.). Nous verrons alors que les LTA ne sont pas clos par déterminisation. La meilleure solution dans ce cas-là est de proposer une sur-approximation de l'automate déterminisé par interprétation abstraite.

Comme troisième et principale contribution, nous décrivons un nouvel algorithme de complétion (utilisant l'*abstraction équationnelle*) qui permet de calculer une sur-approximation de l'ensemble des états accessibles du système, où ces états sont représentés par des termes potentiellement interprétables, et les ensembles d'états par des LTA. Notre algorithme étend l'approche classique de complétion en considérant des systèmes de réécriture conditionnels pouvant contenir des éléments du domaine concret $\mathcal{C}$. Nous verrons également que les conditions de ces systèmes de réécriture impliquent l'utilisation d'un *solveur* pour domaine abstrait durant l'étape de complétion. Comme approximation, nous proposons un nouveau type d'équations permettant de fusionner des termes intégrant des éléments du treillis, afin d'améliorer la terminaison du calcul.

Enfin, nous prouvons que notre algorithme de complétion est correct, dans le sens où il calcule une sur-approximation de l'ensemble des états accessibles. Cette dernière propriété est garantie si chaque étape de complétion est suivie d'une étape d'évaluation. Cette opération, qui va utiliser un opérateur de *widening* (méthode classiquement utilisée en interprétation abstraite), ajoute des termes qui peuvent être perdus lors de l'étape de complétion.

Il est à noter qu'il existe une implémentation décrite au chapitre suivant pour la vérification des programmes $Java$, utilisant le domaine des entiers relatifs en tant que domaine concret et l'ensemble des intervalles en tant que treillis. Mais les LTA et les systèmes de réécriture conditionnels utilisés peuvent utiliser d'autres domaines concrets associés à d'autres treillis. En effet, nous verrons que l'intérêt principal de cet algorithme est qu'il ne dépend pas du treillis utilisé : cet algorithme est *générique* et également *paramétrable*. Le premier paramètre est le domaine concret utilisé, puis le domaine abstrait

(treillis) choisi correspondant est également un paramètre modifiable. On peut également considérer les paramètres liés au treillis lui-même. Ainsi, n'importe quel treillis (respectant certaines conditions décrites dans ce chapitre), peut être *branché* dans cet algorithme afin d'abstraire différents types de domaines concrets, à condition que l'utilisateur fournisse le solveur correspondant.

Travaux liés existants

Ce travail est inspiré par [Le Gall et Jeannet, 2007], où les auteurs proposent d'utiliser des automates de mots à treillis dans le cadre du Model-Checking régulier. Dans cet article, les automates de mots à treillis sont utilisés pour modéliser et vérifier certains protocoles de communication comme les machines symboliques communicantes (*Symbolic Communicating Machine*), faisant intervenir des messages complexes encapsulant des entiers ou des listes d'entiers. Les différences principales avec notre travail sont les suivantes :

(1) nous travaillons avec des termes et non pas des mots,

(2) nous proposons un algorithme de complétion et une technique d'abstraction générique,

(3) nous représentons et calculons les opérations sur les treillis (comme par exemple les opérations arithmétiques dans le cas des entiers), ce qui n'est pas le cas dans leurs automates, et

(4) nous mélangeons les termes non-interprétables et interprétables, alors que leurs automates ne reconnaissent que des mots composés uniquement d'éléments du treillis.

Quelques approches de Model-Checking réguliers peuvent être trouvées dans les travaux suivants : [Abdulla et al., 2007], [Boigelot et al., 2003], [Abdulla et al., 2008], [Bouajjani et al., 2006a]. Cependant, aucune d'entre elles ne permet de capturer les deux dimensions infinies des systèmes complexes d'une manière efficace. D'autres modèles, comme les automates modaux [Bauer et al., 2011] ou les arbres de données (*data trees* : [David et al., 2011, Figueira et Segoufin, 2011, Genest et al., 2010]), considèrent des alphabets infinis, mais n'exploitent pas la structure des treillis comme dans notre travail.

Les automates à treillis évalués de [Kupferman et Lustig, 2007] (ou *lattice-valued automata*), dont les transitions sont étiquetées par les éléments d'un treillis, permettent d'associer chaque *mot* d'un alphabet fini à un élément d'un treillis. Des automates similaires peuvent définir des langages d'*arbres* "flous" (appelés alors *fuzzy tree automata* : [Ésik et Liu, 2007]). Ces automates sont une catégorie particulière des *weighted tree automata* (automates d'arbres dont les transitions sont associées à un opérateur), proposant donc une abstraction des opérateurs de chaque transition.

D'autres méthodes de vérification d'une classe particulière de propriétés des programmes $Java$, avec termes interprétables, peuvent être trouvées dans [Otto et al., 2010]. Dans ces travaux, un outil appelé `AProVE` permet tout d'abord de transformer le $Java$ bytecode du programme en un graphe de terminaison, représentant symboliquement de manière finie tous les chemins d'exécution possibles du programme. Les états de ce graphe de terminaison sont des états *abstraits*, et sont sous forme d'un terme *(Point de programme* $\times$ *Variable locales* $\times$ *Pile des opérateurs* $\times$ *Tas)* et représentent

une abstraction des différents états du programme. Un entier est ainsi abstrait par un intervalle. Le graphe de terminaison généré sera ensuite transformé en un système de réécriture acceptant les entiers (par exemple, la liste $[1, 3, 2]$ est représentée par le terme $IntList(1, IntList(3, IntList(2, null))))$, appelé $ITRS$ (*Integer Term Rewriting System*). Cependant, ces travaux sont orientés sur la preuve de terminaison d'un programme, et ne permettent pas la vérification de propriétés de sûreté. Ils ne gèrent pas non plus la récursivité. De plus, dans ce chapitre, et surtout dans le suivant, nous verrons que nous pouvons représenter directement la sémantique du bytecode $Java$, sans passer par un graphe de terminaison, de plus l'abstraction est faite seulement dans les automates d'arbres, et non pas dans le système de réécriture, qui permet donc de représenter la sémantique concrète du bytecode.

De nombreuses techniques visent à vérifier les programmes contenant de l'arithmétique sur les entiers. Parmi elles, l'*interprétation abstraite* [Cousot et Cousot, 1977] calcule une sur-approximation de l'ensemble des états accessibles, mais requiert l'évaluation complète des expressions arithmétiques, alors que les LTA peuvent traiter les expressions qui sont partiellement évaluées, ce qui peut être utile dans l'analyse interprocédurale. Il existe, en Model-Checking, d'autres manières pour traiter les opérations arithmétiques de manières efficaces, comme par exemple [Leroux, 2008]. Cependant, nous pensons que les LTA permettent d'abstraire de nombreux types de données autres que les entiers, tels que par exemple les chaines de caractères, en intégrant simplement le domaine abstrait adapté (utilisant sa meilleure implémentation disponible). En particulier, les LTA pourraient être utilisés par d'autres techniques de Model-Checking régulier comme [Abdulla et al., 2002, Bouajjani et Touili, 2002], qui ne possèdent pas à ce jour cette capacité.

5.2 Automates d'arbres à treillis (Lattice Tree Automata : *LTA*)

Dans cette section, nous commencerons par expliquer comment ajouter les éléments d'un domaine concret dans un terme (ce qui à notre connaissance a d'abord été défini dans [Kaplan et Choppy, 1989]), et comment passer d'un domaine concret à un domaine abstrait. Nous proposerons ensuite un nouveau type d'automates d'arbres permettant de reconnaitre des termes incluant des éléments d'un treillis et étudierons ses propriétés.

5.2.1 Symboles interprétables et évaluation

Dans ce qui suit, les éléments d'un domaine concret et possiblement infini $\mathcal{C}$ seront représentés par un ensemble de *symboles interprétables* $\mathcal{F}_\bullet$. L'ensemble des symboles est maintenant noté $\mathcal{F} = \mathcal{F}_\circ \cup \mathcal{F}_\bullet$, avec $\mathcal{F}_\circ$ l'ensemble des symboles *non-interprétables*. L'ensemble des symboles *interprétables* $\mathcal{F}_\bullet$ est composé d'éléments de $\mathcal{C}$ (*i.e* $\mathcal{C} \subseteq \mathcal{F}_\bullet$) d'arité 0, et d'un nombre fini d'opérations prédéfinies $op : \mathcal{C}^n \to \mathcal{C}$, avec $op \in \mathcal{F}^n$. On note OP l'ensemble des opérations prédéfinies, et on a alors $\mathcal{F}_\bullet = \mathcal{C} \cup OP$. Par exemple, si $\mathcal{C} = \mathbb{Z}$, alors $\mathcal{F}_\bullet$ peut être $\mathbb{Z} \cup \{+, -, \times\}$ (voir figure 5.2). Les symboles non-interprétables peuvent être assimilés aux symboles fonctionnels usuels, et les symboles interprétables sont des opérations *built-in* du domaine $\mathcal{C}$.

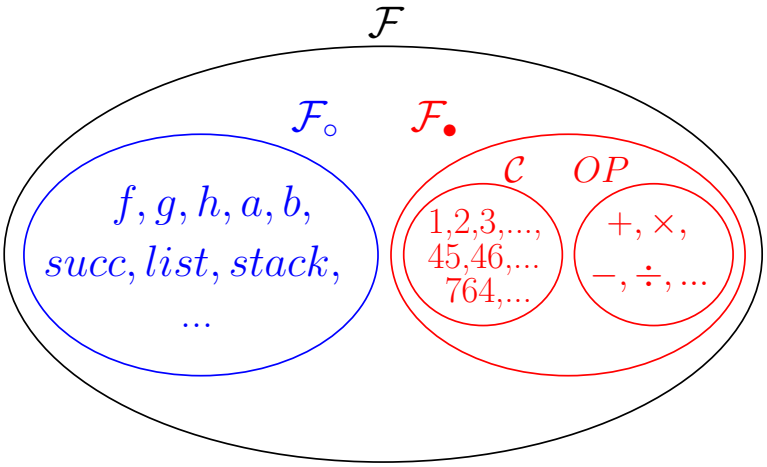


FIGURE 5.2 — $\mathcal{F} = \mathcal{F}_\circ \cup \mathcal{F}_\bullet$.

L'ensemble $\mathcal{T}(\mathcal{F}_\bullet)$ représente l'ensemble des termes construits sur $\mathcal{F}_\bullet$. Les termes de cet ensemble peuvent être évalués en utilisant une fonction d'évaluation $eval : \mathcal{T}(\mathcal{F}_\bullet) \to \mathcal{C}$. Le but de la fonction $eval$ est de simplifier un terme en utilisant les opérations *built-in* du domaine $\mathcal{C}$ (voir exemple 5.1).

La fonction $eval$ s'étend naturellement à $\mathcal{T}(\mathcal{F})$ de la manière suivante :

$$eval(f(t_1, \ldots, t_n)) = \begin{cases} f(eval(t_1), \ldots, eval(t_n)) \text{ si } f \in \mathcal{F}_\circ \text{ ou } \exists i = 1 \ldots n : t_i \notin \mathcal{T}(\mathcal{F}_\bullet) \\ x \in \mathcal{C} \text{ si } f(t_1, \ldots, t_n) \in \mathcal{T}(\mathcal{F}_\bullet). \end{cases}$$

EXEMPLE *5.1*
$eval((1 + (3 \times 3)) - 2) = 8$, et $eval(f(22 - 12, g(4 \times 5))) = f(10, g(20))$. ◀

Les automates d'arbres peuvent être définis de manière à reconnaitre des ensembles de termes interprétables.

EXEMPLE *5.2*
L'ensemble $\{f(1), f(2), f(3), f(4), f(5)\}$ est reconnu par un automate d'arbres $\mathcal{A}_{ex}$ possédant l'ensemble de transitions suivant :

$$\begin{aligned} \Delta_{\mathcal{A}_{ex}} = \{ \quad & 1 \to q,\ 2 \to q, \\ & 3 \to q,\ 4 \to q, \\ & 5 \to q,\ f(q) \to q_f \quad \} \end{aligned}$$

◀

Mais le codage de l'exemple 5.2 n'est pas vraiment efficace, et il est préférable d'avoir des transitions spéciales permettant de manipuler les ensembles d'entiers. Nous voulons donc définir un automate d'arbres avec seulement deux transitions : $\{1, \ldots, 6\} \to q, f(q) \to q_f$. De plus, dans le cas d'une chaîne de réécriture infinie (ex : $f(1) \to f(2) \to f(3) \to \ldots \to f(1000) \to \ldots$), ce codage ne permettra pas non plus de capturer le comportement infini alors que cela serait possible grâce à deux transitions de la forme $\{1, \ldots, +\infty\} \to q, f(q) \to q_f$.

En effet, supposons que l'on souhaite représenter un nombre potentiellement infini d'éléments de notre langage avec un nombre fini de transitions. Il faut que ces transitions reconnaissent un ensemble d'éléments de l'alphabet plutôt que des éléments

uniques. Notre idée principale est de généraliser ce codage et de définir des automates d'arbres possédant des transitions capables de reconnaitre les éléments d'un treillis (les ensembles d'entiers sont les éléments du treillis $2^{\mathbb{Z}}$). Étant donné qu'il est impossible de représenter de manière fini des ensembles d'éléments d'un alphabet infini, nous choisissons ici d'abstraire ces ensembles par des éléments d'un treillis *abstrait*, afin d'améliorer l'efficacité de cette approche (démarche classique dans le domaine de l'*interprétation abstraite*). De plus, le Model-Checking régulier sur les arbres (RTMC) nécessite seulement une sur-approximation de l'ensemble des états accessibles. Nous avons donc choisi d'avoir des transitions spéciales permettant de reconnaitre les éléments d'un *treillis abstrait* $(\Lambda, \sqsubseteq)$, comme par exemple le treillis des intervalles.

EXEMPLE *5.3*
Dans l'exemple 5.2, si l'on définit l'automate précédent $\mathcal{A}_{ex}$ en utilisant le treillis des intervalles sur les entiers, on obtient l'ensemble de transitions suivant : $\Delta_{\mathcal{A}_{ex}} = \{[1,6] \to q, f(q) \to q_f\}$. ◀

Pour chaque opération *built-in* $op \in OP$ définie sur $\mathcal{C}$, l'opération correspondante $op^{\#} \in OP^{\#}$ est définie sur Λ. Comme $\mathcal{F}_{\bullet} = \mathcal{C} \cup OP$, l'ensemble des symboles *abstraits* est défini par $\mathcal{F}_{\bullet}^{\#} = \Lambda \cup OP^{\#} \cup \{\sqcup, \sqcap\}$. Les opérations $\sqcup$ et $\sqcap$, représentant respectivement le lub et le glb (voir section 2.6), sont d'arité 2 et les opérations $op^{\#}$ et op sont de même arité (voir figure 5.3). Par exemple, soit I l'ensemble des intervalles dont les bornes appartiennent à $\mathbb{Z} \cup \{-\infty, +\infty\}$. L'ensemble $\mathcal{F}_{\bullet} = \mathbb{Z} \cup \{+, -\}$ peut être abstrait par $\mathcal{F}_{\bullet}^{\#} = I \cup \{+^{\#}, -^{\#}, \sqcup, \sqcap\}$. Les termes contenant des opérateurs étendus pour le domaine abstrait doivent être évalués, comme expliqué au début de cette section pour des éléments d'un domaine concret. Si il y a une connexion de Galois entre le domaine concret et le domaine abstrait, $eval^{\#} : \mathcal{T}(\mathcal{F}_{\bullet}^{\#}) \mapsto \Lambda$ est la meilleure approximation de la fonction $eval$ respectant cette connexion de Galois.

EXEMPLE *5.4* (Fonction $eval^{\#}$)
Il y a une connexion de Galois entre $(2^{\mathbb{Z}}, \subseteq)$ et le treillis des intervalles $(I, \sqsubseteq)$. La fonction $eval^{\#}$ est définie par :

- $eval^{\#}(i) = i$ pour tout intervalle i, donc $eval^{\#}(\bot) =]+\infty, -\infty[$ et $eval^{\#}(\top) =]-\infty, +\infty[$,
- Pour chaque $f \in \{+^{\#}, -^{\#}, \sqcup, \sqcap\}$, si $eval^{\#}(i_1) = [a,b]$ et $eval^{\#}(i_2) = [c,d]$, alors $eval^{\#}(f(i_1, i_2))$ est définie par :
  - $eval^{\#}([a,b] \sqcup [c,d]) = [min(a,c), max(b,d)]$
  - $eval^{\#}([a,b] \sqcap [c,d]) = [max(a,c), min(b,d)]$ si $max(a,c) \leq min(b,d)$, sinon $eval^{\#}([a,b] \sqcap [c,d]) = \bot$,
  - $eval^{\#}([a,b] +^{\#} [c,d]) = [a+c, b+d]$
  - $eval^{\#}([a,b] -^{\#} [c,d]) = [a-d, b-c]$.

◀

5.2.2 Définition et sémantique

Les automates d'arbres à treillis –notés LTA (Lattice Tree Automata) sur le reste du chapitre pour plus de simplicité– étendent les automates d'arbres classiques afin de reconnaitre des termes définis sur $\mathcal{F}_{\circ} \cup \mathcal{F}_{\bullet}^{\#}$.

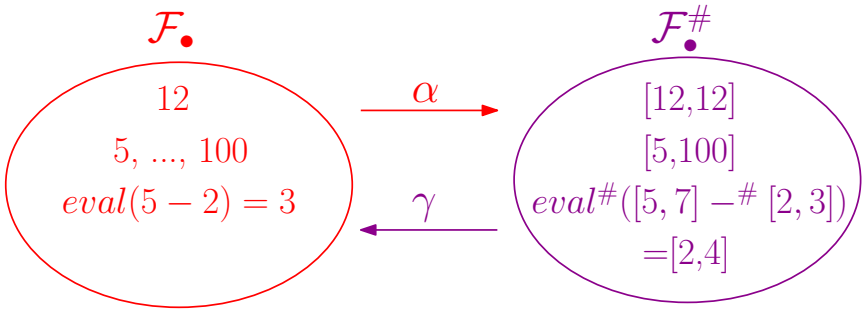


FIGURE 5.3 — $\mathcal{F}_\bullet$ et sa correspondance abstraite $\mathcal{F}_\bullet^\#$ dans le cas des entiers abstraits par des ensembles d'intervalles.

DÉFINITION 5.2.1 (Automate d'arbres à treillis (LTA))
Un automate d'arbres ascendant non-déterministe à treillis (ou, de manière plus concise, un automate d'arbres à treillis, ou lattice tree automaton : LTA) est un tuple $\mathcal{A} = \langle \mathcal{F}_\circ \cup \mathcal{F}_\bullet^\#, \mathcal{Q}, \mathcal{Q}_F, \Delta \rangle$, avec $\mathcal{F}_\circ$ un ensemble de symboles non-interprétables et $\mathcal{F}_\bullet^\# = \Lambda \cup OP^\#$ un ensemble de symboles interprétables, $\mathcal{Q}$ un ensemble d'états et $\mathcal{Q}_F$ un ensemble d'états finaux, $\mathcal{Q}_F \subseteq \mathcal{Q}$, et Δ un ensemble de transitions normalisées.

Nous supposons que le treillis utilisé est *unidimensionnel*, i.e. ne permettant d'abstraire qu'une seule valeur, par opposition aux treillis relationnels tels que les polyèdres, permettant d'abstraire un ensemble de valeurs en établissant des relations entre plusieurs variables. En effet, étant donné qu'un élément du treillis est reconnu par un état de l'automate, il ne peut abstraire qu'une seule valeur. Nous parlerons de l'intégration de treillis *multidimensionnels* (ou *relationnels*) dans les perspectives de cette thèse (chapitre Conclusions et perspectives). De plus, nous supposons que ce treillis abstrait est *atomique* (voir définition 2.6.3), pour certaines raisons qui seront expliquées à la fin de cette sous-section.

L'ensemble des *λ-transitions* est défini par $\Delta_\Lambda = \{\lambda \to q \mid \lambda \to q \in \Delta \wedge \lambda \neq \bot \wedge \lambda \in \Lambda\}$. L'ensemble des *transitions closes* est l'ensemble des autres transitions de l'automate, et cet ensemble est formellement défini par $\Delta_G = \{f(q_1, \ldots, q_n) \to q \mid f(q_1, \ldots, q_n) \to q \in \Delta \wedge q, q_1, \ldots, q_n \in Q\}$.

L'ordre partiel $\sqsubseteq$ du treillis Λ est étendu sur l'ensemble $\mathcal{T}(\mathcal{F})$ de la manière suivante :

DÉFINITION 5.2.2 ($\sqsubseteq$ sur $\mathcal{T}(\mathcal{F})$)
Soit $s, t \in \mathcal{T}(\mathcal{F})$, $s \sqsubseteq t$ si et seulement si :

1. $eval(s) \sqsubseteq eval(t)$ *(si s et t appartiennent à $\mathcal{T}(\mathcal{F}_\bullet^\#)$),*
2. $s = f(s_1, \ldots, s_n)$, $t = f(t_1, \ldots, t_n)$, $f \in \mathcal{F}_\circ^n$ *et* $s_1 \sqsubseteq t_1 \wedge \ldots \wedge s_n \sqsubseteq t_n$.

EXEMPLE *5.5*
$f(g(a, [1,5]) \sqsubseteq f(g(a, [0,8])$, et $h([0,4] +^\# [2,6]) \sqsubseteq h([1,3] +^\# [1,9])$. ◀

Dans ce qui suit, le symbole $\#$ va être omis quand sa présence est évidente vis-à-vis du contexte. Nous allons maintenant définir la relation de transition induite par les LTA (automates d'arbres à treillis) et leur langage reconnu. La différence avec un automate

d'arbres classique réside dans le fait qu'un terme t est reconnu par un LTA seulement si l'évaluation de celui-ci ($eval(t)$) peut être réduite dans l'automate.

DÉFINITION 5.2.3 ($t_1 \to_{\mathcal{A}} t_2$ pour les LTA)
Soit $t_1, t_2 \in \mathcal{T}(\mathcal{F} \cup \mathcal{Q})$. $t_1 \to_{\mathcal{A}} t_2$ si pour toute position $p \in Pos(t_1)$:
- *si $t_1|_p \in \mathcal{T}(\mathcal{F}_\bullet^\#)$, il existe une transition $\lambda \to q \in \Delta$ telle que $eval(t_1|_p) \sqsubseteq \lambda$ et $t_2 = t_1[q]_p$*
- *si $t_1|_p = q$ avec $q \in \mathcal{Q}$, il existe une ϵ-transition $q \to q' \in \Delta$, avec $q' \in \mathcal{Q}$ tel que $t_2 = t_1[q']_p$*
- *si $t_1|_p = f(s_1, \ldots, s_n)$ avec $f \in \mathcal{F}^n$ et $s_1, \ldots s_n \in \mathcal{T}(\mathcal{F} \cup \mathcal{Q})$, $\exists s'_i \in \mathcal{T}(\mathcal{F} \cup \mathcal{Q})$ tel que $s_i \to_{\mathcal{A}} s'_i$ et $t_2 = t_1[f(s_1, \ldots, s_{i-1}, s'_i, s_{i+1}, \ldots, s_n)]_p$*
- *si $t_1|_p = f(q_1, \ldots, q_n)$ avec $f \in \mathcal{F}^n$ et $q_1, \ldots q_n \in \mathcal{Q}$, il existe une transition $f(q_1, \ldots, q_n) \to q \in \Delta$ telle que $t_2 = t_1[q]_p$.*

$\to_{\mathcal{A}}^*$ est la fermeture réflexive transitive de $\to_{\mathcal{A}}$. Un terme t_1 peut se réécrire en t_2 dans l'automate si $t_1 \to_{\mathcal{A}}^* t_2$.

On note $\mathcal{T}(\mathcal{F}, Atoms(\Lambda))$ l'ensemble des termes clos contruits sur $(\mathcal{F} \backslash \Lambda) \cup Atoms(\Lambda)$. Les LTA reconnaissent un langage d'arbre sur $\mathcal{T}(\mathcal{F}, Atoms(\Lambda))$. Nous verrons à la fin de cette sous-section pourquoi le langage est défini sur les atomes du treillis.

DÉFINITION 5.2.4 (Langage reconnu)
Le langage d'arbres reconnu par $\mathcal{A}$ sur un état q est défini par :

$$\mathcal{L}(\mathcal{A}, q) = \{t \in \mathcal{T}(\mathcal{F}, Atoms(\Lambda)) \mid \exists\, t' \text{ tel que } t \sqsubseteq t' \text{ et } t' \to_{\mathcal{A}}^* q\}.$$

Le langage reconnu par $\mathcal{A}$ est $\mathcal{L}(\mathcal{A}) = \bigcup_{q \in \mathcal{Q}_f} \mathcal{L}(\mathcal{A}, q)$.

EXEMPLE *5.6* ($\to_{\mathcal{A}}^*$, langage reconnu)
Soit $\mathcal{A} = \langle \mathcal{F} = \mathcal{F}_\circ \cup \mathcal{F}_\bullet^\#, \mathcal{Q}, \mathcal{Q}_f, \Delta \rangle$ un LTA avec $\Delta = \{[0,4] \to q_1, f(q_1) \to q_2\}$, et $\mathcal{Q}_f = \{q_2\}$ est le seul état final. On a alors par exemple :
$f([1,4]) \to^* q_2$ et $f([0,2]) \to^* q_2$, et le langage reconnu par $\mathcal{A}$ est le suivant :
$\mathcal{L}(\mathcal{A}) = \mathcal{L}(\mathcal{A}, q_2) = \{f([0,0]), f([1,1]), \ldots, f([4,4])\}$. ◀

Remarque (Treillis atomique et langage sur les atomes). Les raisons pour lesquelles nous considérons seulement les treillis atomiques découlent de notre but : représenter de manière concise et efficace les langages sur alphabet infinis. Dans ce cadre, chaque atome du treillis est une lettre de l'alphabet. L'existence d'une partition finie des atomes permet de représenter de manière finie un ensemble potentiellement infini de lettres de l'alphabet, avec au plus autant de λ-transitions qu'il y a de classes dans la partition. Nous aurons besoin de cette propriété pour définir l'opération de *déterminisation*, comme nous allons le voir en section suivante.

L'existence d'une partition finie est également nécessaire pour l'algorithme de filtrage. Lors de l'algorithme de complétion, nous verrons en section 5.4 que l'algorithme de filtrage, et notamment la résolution des paires critiques, nécessite l'utilisation d'un *solveur*. En effet, les systèmes de réécriture sont conditionnels, et une règle s'applique donc sous certaines conditions que devront respecter les nouvelles transitions ajoutées durant la résolution des paires critiques. Si sous une substitution σ retournée par l'algorithme de filtrage se trouve un élément du treillis, alors, si cet élément est contraint dans la règle de réécriture, la nouvelle transition doit respecter cette contrainte. Pour cela,

l'élément du treillis en question doit possiblement être restreint. Prenons par exemple la règle de réécriture $f(x) \to g(x) \Leftarrow x > 1$ et prenons $\sigma = \{x \mapsto q\}$ une substitution obtenue. Imaginons maintenant qu'il y ait une transition $[0,3] \to q$ dans l'automate. Alors l'intervalle $[0,3]$ doit respecter la condition $x \geq 1$. Comme ce n'est pas le cas, il faut alors restreindre l'intervalle, pour obtenir l'intervalle $[2,3]$ respectant la solution. Une telle restriction ne serait pas possible si le treillis n'était pas atomique. En effet, si l'on prend le treillis comportant uniquement l'élément 0, ce treillis se représente alors de cette manière : $\bot$——— 0 ———$\top$, et n'est pas atomique. Dans ce cas, la contrainte $x \geq 1$ renvoie alors l'élément $\top$ qui contient l'élément 0 et ne restreint donc pas l'élément contraint.

En outre, cette sémantique évite d'avoir à considérer tous les éléments qui sont plus petit que l'étiquette d'une transition sans pour autant être des atomes. En effet, sur l'exemple précédent, plutôt que d'inclure toutes les combinaisons possibles d'intervalles reconnaissant les entiers de 1 à 4 dans le langage de $\mathcal{A}$ (*i.e.* $\mathcal{L}(\mathcal{A}) = \mathcal{L}(\mathcal{A}, q_2) = \{f([0,0]), f([0,1]),\ f([0,2],\ \ldots, f([1,1]),\ f([1,2]),\ \ldots, f([4,4])\}$), il est préférable d'inclure dans le langage uniquement les atomes pour plus de concision et de clarté. De plus, on peut construire tous les intervalles reconnus grâce à l'ensemble des atomes. Par exemple, $[2,4] = [2,2] \cup [3,3] \cup [4,4]$, donc si $f([2,2]$, $f([3,3])$ et $f([4,4])$ appartiennent à $\mathcal{L}(\mathcal{A})$, alors $f([2,4])$ est bien reconnu par l'automate $\mathcal{A}$.

5.3 Opérations sur les LTA

Dans cette, section, nous étudions et définissons diverses opérations possibles sur les LTA qui sont utilisées de manière classique en Model-Checking.

5.3.1 Opérations booléennes

La plupart des algorithmes définis pour les opérations booléennes sont des adaptations de ceux définis pour les automates d'arbres classiques (voir [Comon et al., 2007]).

Les LTA sont clos par union et intersection, et nous allons rapidement expliquer comment ces deux opérations $\cup$ et $\cap$ peuvent être appliquées entre deux LTA $\mathcal{A} = \langle \mathcal{F}, \mathcal{Q}, \mathcal{Q}_f, \Delta \rangle$ et $\mathcal{A}' = \langle \mathcal{F}, \mathcal{Q}', \mathcal{Q}'_f, \Delta' \rangle$:

- $\mathcal{A} \cup \mathcal{A}' = \langle \mathcal{F}, \mathcal{Q} \cup \mathcal{Q}', \mathcal{Q}_f \cup \mathcal{Q}'_f, \Delta \cup \Delta' \rangle$ en supposant que les ensembles $\mathcal{Q}$ et $\mathcal{Q}'$ sont disjoints.
- Soient Λ le treillis tel que $\Lambda \subseteq \mathcal{F}$ et $\sqcap$ le glb associé, $\mathcal{A} \cap \mathcal{A}'$ est le LTA $\mathcal{A} \cap \mathcal{A}' = \langle \mathcal{F}, \mathcal{Q} \times Q', \mathcal{Q}_f \times \mathcal{Q}'_f, \Delta_\cap \rangle$ où les transitions de $\Delta_\cap$ sont définies par les règles suivantes :
$$\frac{\lambda \to q \in \Delta \quad \lambda' \to q' \in \Delta' \quad \lambda \sqcap \lambda' \neq \bot}{\lambda \sqcap \lambda' \to (q, q')}$$
et
$$\frac{f(q_1, \ldots, q_n) \to q \in \Delta \quad f(q'_1, \ldots, q'_n) \to q' \in \Delta'}{f((q_1, q'_1), \ldots, (q_n, q'_n)) \to (q, q')}$$

Le Model-Checking régulier sur arbres requiert aussi l'application d'une opération de complément et de décision du vide. Le LTA complément est obtenu en inversant les états finaux et non-finaux, mais cet algorithme fonctionne si l'automate donné en entrée est déterministe. Un algorithme de déterminisation sera donné en sous-section suivante.

D'autre part, pour décider si le langage décrit par un LTA est vide ou non, il suffit d'observer qu'un automate accepte au moins un terme si et seulement si au moins un état final est accessible. Un automate réduit est un automate délesté de ces états inaccessibles. Le langage d'un automate réduit est vide si et seulement si l'ensemble des états finaux est vide. Pour l'opération de décision du vide, la première étape est donc de réduire le LTA, *i.e.* supprimer l'ensemble des états inaccessibles. Rappelons l'algorithme de réduction :

Algorithme de réduction
Entrée : LTA $\mathcal{A} = \langle \mathcal{F}, \mathcal{Q}, \mathcal{Q}_f, \Delta \rangle$
Début
Marked:=$\emptyset$
/* Marked est l'ensemble des états accessibles */
Répéter
si ($\exists a \in \mathcal{F}^0 = \mathcal{F}^0_\circ \cup \mathcal{F}^{\#^0}_\bullet$ tel que $a \to q \in \Delta$
ou $\exists f \in \mathcal{F}^n = \mathcal{F}^n_\circ \cup \mathcal{F}^{\#^n}_\bullet$ tel que $f(q_1, \ldots, q_n) \to q \in \Delta$)
et ($q \notin Marked$)
avec $q_1, \ldots, q_n \in Marked$
alors $Marked := Marked \cup \{q\}$
Jusqu'à ce qu'il n'y ait plus d'états à ajouter à l'ensemble *Marked*
$\mathcal{Q}_r := Marked$
$\mathcal{Q}_{r_f} := \mathcal{Q}_f \cap Marked$
$\Delta_r := \{f(q_1, \ldots, q_n) \to q \in \Delta | q, q_1, \ldots, q_n \in Marked\}$
Sortie : Le LTA réduit : $\mathcal{A}_r = \langle \mathcal{F}, \mathcal{Q}_r, \mathcal{Q}_{r_f}, \Delta_r \rangle$
Fin

Ensuite, soit $\mathcal{A}$ un LTA et $\mathcal{A}_r = \langle \mathcal{F}, \mathcal{Q}_r, \mathcal{Q}_{r_f}, \Delta_r \rangle$ le LTA *réduit* correspondant, $\mathcal{L}(\mathcal{A})$ est vide si et seulement si $\mathcal{Q}_{r_f} = \emptyset$.

EXEMPLE *5.7* (Décision du vide)
Soit $\mathcal{A}$ le LTA possédant l'ensemble de transitions suivant :
$\Delta = \{[1, 2] \to q_1, f(q_2) \to q_3\}$,
avec q_3 l'unique état final. L'ensemble des états accessibles est $Marked = \{q_1\}$. q_2 et q_3 sont en effet inaccessibles. Il n'y a donc aucun état final accessible, le langage de cet automate est donc vide.

Par contre, si l'on prend un LTA possédant les transitions suivantes :
$\Delta = \{ \quad [1, 2] \to q_1, f(q_2) \to q_3,$
$\quad [3, 4] \to q_4, f(q_4) \to q_5\}$
avec q_3 et q_5 les états finaux, alors $Marked = \{q_1, q_4, q_5\}$ et q_5 est un état final. Le langage de cet automate n'est donc pas vide. ◀

Soit $\mathcal{A}_1$, $\mathcal{A}_2$ deux LTA. En supposant qu'il existe un automate $\overline{\mathcal{A}_2}$ reconnaissant le complémentaire de $\mathcal{L}(\mathcal{A}_2)$, alors on a $\mathcal{L}(\mathcal{A}_1) \subseteq \mathcal{L}(\mathcal{A}_2) \Leftrightarrow \mathcal{L}(\mathcal{A}_1 \cap \overline{\mathcal{A}_2}) = \emptyset$.

Le calcul classique du complément, et par extension la méthode classique de décision de l'inclusion, sont prouvées correctes pour des automates déterministes. Cependant, en adaptant la preuve des automates de *mots* à treillis donnée dans [Le Gall et Jeannet, 2007], on peut montrer que les LTA ne sont pas clos par déterminisation.

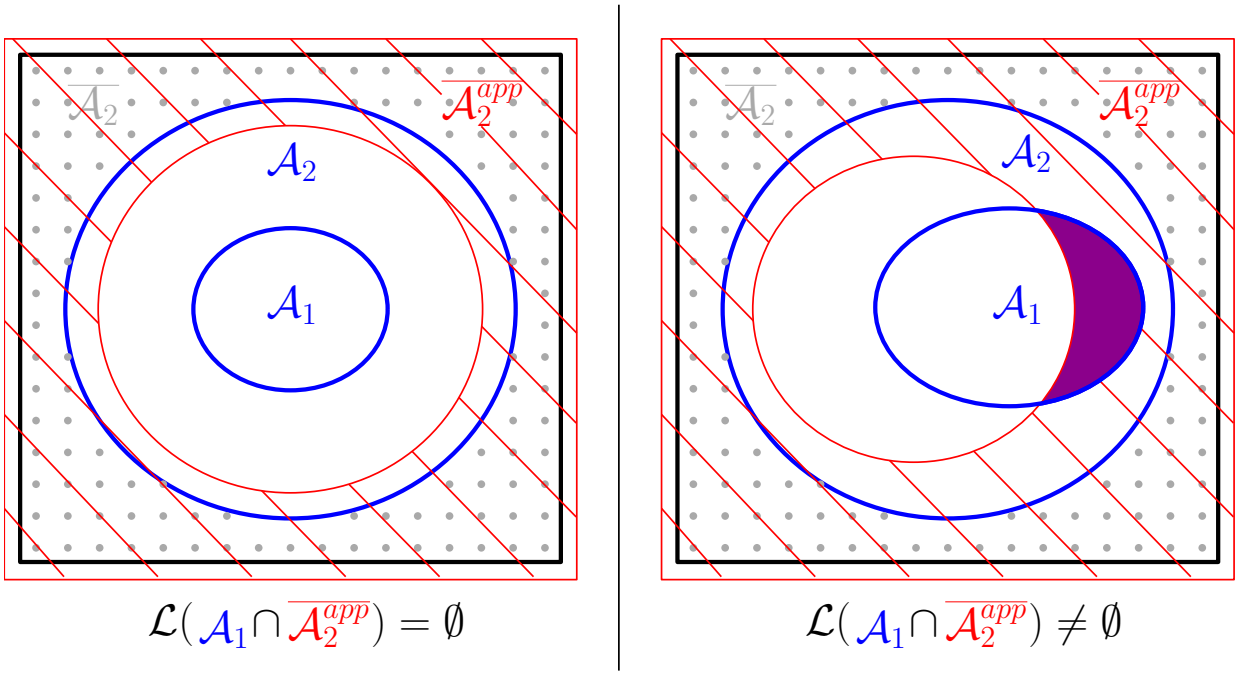


FIGURE 5.4 — Sur-approximation du complément pour le test d'inclusion.

Dans la sous-section suivante, nous proposons un algorithme qui calcule une sur-approximation déterministe pour un LTA donné. Cet algorithme, qui étend celui de [Le Gall et Jeannet, 2007], repose sur une fonction de partitionnement qui peut être raffinée pour rendre la sur-approximation plus précise. Cette déterminisation permet d'avoir une sur-approximation du complément et donc de donner seulement un critère de réponse pour l'inclusion. En effet, comme nous le voyons sur la figure 5.4, soit $\overline{\mathcal{A}_2^{app}}$ une sur-approximation du complément de $\mathcal{A}_2$, si l'on a $\mathcal{L}(\mathcal{A}_1 \cap \overline{\mathcal{A}_2^{app}}) = \emptyset$, on peut conclure que $\mathcal{L}(\mathcal{A}_1) \subseteq \mathcal{L}(\mathcal{A}_2)$. Cependant, si $\mathcal{L}(\mathcal{A}_1 \cap \overline{\mathcal{A}_2^{app}}) \neq \emptyset$, il n'est pas possible de savoir si l'on est en présence d'un vrai ou d'un faux contre-exemple.

Remarque. Il est important noter que nous décrivons ici l'algorithme de déterminisation afin de pouvoir utiliser les LTA avec les méthodes classiques de Model-Checking qui nécessitent l'inclusion. Cependant dans ce chapitre, nous n'avons *nul besoin* d'utiliser ces algorithmes pour calculer l'ensemble des accessibles. **Seule la décision du vide** est nécessaire à l'algorithme de complétion.

5.3.2 Déterminisation

Comme nous allons le voir, un LTA $\mathcal{A} = \langle \mathcal{F}, \mathcal{Q}, \mathcal{Q}_f, \Delta \rangle$ est *déterministe* s'il n'y a aucune transition $f(q_1, \ldots, q_n) \to q$, $f(q_1, \ldots, q_n) \to q'$ dans Δ telle que $q \neq q'$, avec $f \in \mathcal{F}_n$, et aucune transition $\lambda_1 \to q$, $\lambda_2 \to q'$ telle que $q \neq q'$ et $\lambda_1 \sqcap \lambda_2 \neq \bot$, avec $\lambda_1, \lambda_2 \in \Lambda$. Par exemple, si $\Delta = \{[1,3] \to q_1, [2,5] \to q_2\}$, alors $\mathcal{A}$ n'est pas déterministe.

Déterminiser un LTA par la méthode classique nécessite le calcul des compléments des éléments du treillis. Considérons le LTA $\mathcal{A}$ possédant les transitions suivantes : $[-3,2] \to q_1$ et $[1,6] \to q_2$. Alors le LTA déterministe correspondant devrait avoir les transitions suivantes : $[-3,1[\to q_1$, $[1,2] \to \{q_1, q_2\}$ et $]2,6] \to q_2$. Pour produire de telles transitions, on doit calculer $[-3,2] \sqcap [1,6] = [1,2]$, puis $[-3,2] \setminus [1,2]$ et $[1,6] \setminus [1,2]$. Mais

il n'est pas toujours possible de calculer le complément. En effet certains treillis, comme celui des intervalles, ne sont pas clos par complément.

EXEMPLE *5.8*
Supposons que l'on souhaite déterminiser les deux transitions suivantes : $[0, 12] \to q_1$ et $[2, 6] \to q_2$. Il faut alors calculer $[0, 12] \sqcap [2, 6] = [2, 6]$, ainsi que le complément $[0, 12] \setminus [2, 6]$. Or, $[0, 12] \setminus [2, 6] = [0, 2[\sqcup]6, 12] = [0, 12]$, qui est une approximation du complément et non pas le résultat exact. Le treillis des intervalles n'est donc pas clos par complément. ◀

Par conséquent, la déterminisation d'un LTA ne préserve pas le langage reconnu. La solution proposée dans [Le Gall et Jeannet, 2007] pour les automates de *mots* à treillis est d'utiliser une partition finie du treillis Λ, qui va déterminer quand deux transitions doivent être fusionnées en utilisant l'opérateur lub.

La fusion des transitions peut entrainer une sur-approximation contrôlée par la finesse de la partition.

LTA partitionné (PLTA). Π est une *partition* d'un treillis atomique Λ si $\Pi \subseteq 2^{\Lambda}$ et $\forall \pi \in \Pi$, $\pi \neq \emptyset$, $\forall \pi_1, \pi_2 \in \Pi$, $\pi_1 \sqcap \pi_2 = \bot$, et $\forall a \in Atoms(\Lambda), \exists \pi \in \Pi : a \sqsubseteq \pi$.

EXEMPLE *5.9*
Si Λ est le treillis des intervalles, on peut avoir la partition $\Pi = \{]-\infty, 0[, [0, 0],]0, +\infty[\}$. ◀

DÉFINITION 5.3.1 (LTA partitionné (PLTA).)
Un LTA partitionné $\mathcal{A}$ (noté PLTA pour la suite pour plus de simplicité) est un automate $\mathcal{A} = \langle \Pi, \mathcal{F}, \mathcal{Q}, \mathcal{Q}_f, \Delta \rangle$ équipé d'une partition Π, telle que pour toute λ-transition $\lambda \to q \in \Delta$, $\exists \pi \in \Pi$ telle que $\lambda \sqsubseteq \pi$.

Un PLTA (LTA partitionné) est fusionné si $\lambda_1 \to q$, $\lambda_2 \to q \in \Delta \wedge \lambda_1 \sqsubseteq \pi_1 \wedge \lambda_2 \sqsubseteq \pi_2 \implies \pi_1 \sqcap \pi_2 = \bot$, avec $\lambda_1, \lambda_2 \in \Lambda$ et $\pi_1, \pi_2 \in \Pi$.

EXEMPLE *5.10*
Soit un LTA $\mathcal{A}$ possédant les transitions suivantes :
$\Delta_{\mathcal{A}} = \{ \quad [-3, 0] \to q_1, [-5, -2] \to q_2,$
$\qquad [-1, 4] \to q_4 \}.$

Imaginons que l'on ait une partition $\Pi = \{] - \infty, 0[, [0, 0],]0, +\infty[\}$, alors ce LTA n'est pas partitionné, car il existe une λ-transition $\lambda \to q$ telle qu'il n'existe *aucun* $\pi \in \Pi$ vérifiant $\lambda \sqsubseteq \pi$. En effet, la transition $[-1, 4] \to q_4$ appartient à cet automate, et $[-1, 4] \not\sqsubseteq] - \infty, 0[$, $[-1, 4] \not\sqsubseteq [0, 0]$ et $[-1, 4] \not\sqsubseteq]0, +\infty[$.

Si $\Pi = \{] - \infty, 0[, [0, 0],]0, +\infty[\}$, un LTA *partitionné* ($P$LTA) $\mathcal{A}_p$ peut posséder les transitions suivantes :
$\Delta_{\mathcal{A}_p} = \{ \quad [-3, -1] \to q_1, [-5, -2] \to q_2,$
$\qquad [3, 4] \to q_4 \}.$
Ce PLTA n'est pas fusionné à cause des deux λ-transitions $[-3, -1] \to q_1$ et $[-5, -2] \to q_2$, car $[-3, -1]$ et $[-5, -2]$ sont dans la même partition. L'automate fusionné correspondant possèdera la transition suivante : $[-5, -1] \to q_{1,2}$, à la place des deux transitions mentionnées précédemment. ◀

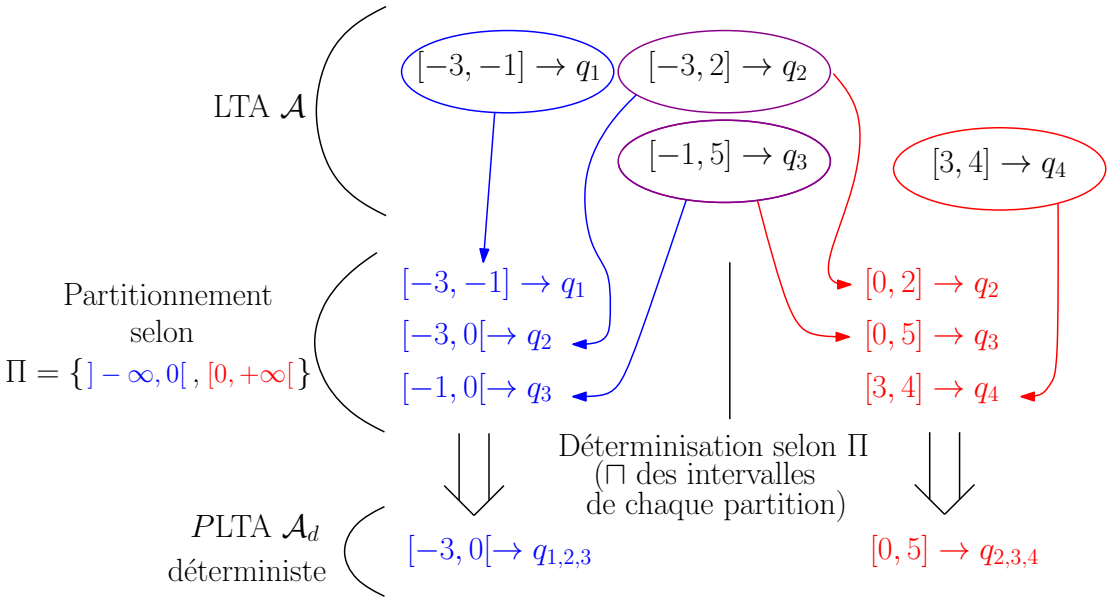


FIGURE 5.5 — Partionner, puis déterminiser un LTA.

Chaque LTA $\mathcal{A}$ peut être transformé en un PLTA $\mathcal{A}_p$ de la manière suivante : Soit Π une partition. Pour chaque λ-transition $\lambda \to q \in \mathcal{A}$, si $\exists \pi_1, \ldots, \pi_n \in \Pi$ telles que $\lambda \sqcap \pi_1 \neq bot, \ldots, \lambda \sqcap \pi_n \neq \bot$, avec $\pi_i \neq \pi_j \forall i, j \in [1, n]$, la transition $\lambda \to q$ sera remplacée par n transitions $\lambda \sqcap \pi_1 \to q, \ldots, \lambda \sqcap \pi_n \to q$ dans $\mathcal{A}_p$ (voir figure 5.5).

EXEMPLE *5.11*
Soit $\mathcal{A} = \langle \mathcal{F}, \mathcal{Q}, \mathcal{Q}_f, \Delta \rangle$ un LTA tel que
$\Delta = \{ \quad [3,4] \to q_1, [-3,2] \to q_2,$
$\quad f(q_1, q_2) \to q_f\}$,
et $\Pi = \{]-\infty, 0[, [0,0],]0, +\infty[\}$ une partition. Alors le PLTA correspondant est $\mathcal{A}_p = \langle \mathcal{F}, \mathcal{Q}, \mathcal{Q}_f, \Delta_p \rangle$, avec
$\Delta_p = \{ \quad [3,4] \to q_1, [-3,0[\to q_2,$
$\quad [0,0] \to q_2,]0,2] \to q_2,$
$\quad f(q_1, q_2) \to q_f\}$.

◀

Deux λ-transitions $\lambda_1 \to q$ et $\lambda_2 \to q$ d'un PLTA ne peuvent être fusionnées si λ_1 et λ_2 appartiennent à différents éléments de la partition, alors qu'elles peuvent être fusionnées dans le cas contraire.

Nous allons maintenant montrer qu'un LTA et son PLTA associé reconnaissent le même langage. Ceci est garanti par l'atomicité du treillis. En effet, lors du partitonnement de l'automate, lorsqu'on découpe une λ-transition selon une partition donnée, les propriétés de l'atomicité permettent de de garantir qu'aucun atome ne sera perdu lors du découpage, et de ce fait, que le langage de l'automate sera conservé, comme nous allons le voir dans la preuve du théorème suivant.

PROPOSITION 5.3.2 (Équivalence entre LTA et PLTA)
Soit un LTA $\mathcal{A} = \langle \mathcal{F}, \mathcal{Q}, \mathcal{Q}_f, \Delta \rangle$ et une partition Π, il existe un PLTA $\mathcal{A}' = \langle \Pi, \mathcal{F}, \mathcal{Q}, \mathcal{Q}_f, \Delta' \rangle$ reconnaissant le même langage (car on suppose le treillis comme étant atomique).

Démonstration. Soient $Atoms(\Lambda)$ l'ensemble des atomes du treillis Λ, $\Pi = \{\pi_1, \ldots, \pi_n\}$ une partition, et un LTA $\mathcal{A} = \langle \mathcal{F}, \mathcal{Q}, \mathcal{Q}_f, \Delta \rangle$. Alors pour toute λ-transition $\lambda \to q$ de Δ, on a $\mathcal{L}(q) = \{a \in Atoms(\Lambda) \mid a \sqsubseteq \lambda\}$ de par la définition 5.2.4. Et pour tout $a \in \mathcal{L}(q)$, il existe un unique $i \in [1, n]$ tel que $a \sqsubseteq \pi_i$, donc par définition du glb $\sqcap$, on a $a \sqsubseteq \lambda \sqcap \pi_i$.

Or le PLTA $\mathcal{A}'$ est obtenu depuis $\mathcal{A}$ en remplaçant chaque λ-transition $\lambda \to q \in \Delta$ par au plus n transitions $\lambda_i \to q$ avec $\lambda_i = \lambda \sqcap \pi_i$, $\pi_i \in \Pi$, tel que $\bigsqcup \lambda_i = \lambda$.

Donc, pour toute λ-transition $\lambda \to q \in \Delta$, tout atome $a \in (q)$ est préservé et conservé dans le langage de l'automate partitionné.

Inversement, pour tout atome $a \sqsubseteq \lambda \sqcap \pi_i$ reconnu par un automate partitionné, pour un certain $i \in [1, n]$, on a alors $a \sqsubseteq \lambda$ et a est donc également reconnu par le LTA de départ non partitionné. Le langage du LTA et celui du PLTA associé sont donc équivalents. □

Chaque PLTA $\mathcal{A} = \langle \Pi, \mathcal{F}, \mathcal{Q}, \mathcal{Q}_f, \Delta \rangle$ peut être transformé en un PLTA *fusionné* $\mathcal{A}_m = \langle \Pi, \mathcal{F}, \mathcal{Q}, \mathcal{Q}_f, \Delta m \rangle$ tel que $\mathcal{L}(\mathcal{A}) \subseteq \mathcal{L}(\mathcal{A}_m)$, en fusionnant les transitions comme suit :

$$\frac{q \in \mathcal{Q} \quad \pi \in \Pi \quad \lambda_m = \bigsqcup\{\lambda \sqcap \pi, \lambda \in \Lambda | \lambda \to q \in \Delta\}}{\lambda_m \to q \in \Delta_m}$$

EXEMPLE *5.12*
Si $\mathcal{A} = \langle \Pi, \mathcal{F}, \mathcal{Q}, \mathcal{Q}_f, \Delta \rangle$, avec $\Pi =]-\infty, 0[, [0, +\infty$ et

$$\begin{aligned} \Delta = \quad & \{[0,2] \to q_1, [5,8] \to q_2, \\ & [-3,-2] \to q_3, [-4,-1] \to q_4, \\ h(q_1, q_2, q_3, q_4) \to q_f\}, & \end{aligned}$$

alors l'automate fusionné $\mathcal{A}_m = \langle \Pi, \mathcal{F}, \mathcal{Q}, \mathcal{Q}_f, \Delta m \rangle$ correspondant à $\mathcal{A}$ possède les transitions suivantes :

$$\begin{aligned} \Delta_m = \quad & \{[0,8] \to q_{1,2}, [-4,-1] \to q_{3,4}, \\ & h(q_{1,2}, q_{1,2}, q_{3,4}, q_{3,4}) \to q_f\}. \end{aligned}$$

◀

Nous sommes maintenant prêt à décrire l'algorithme de déterminisation. La déterminisation d'un PLTA $\mathcal{A}$ qui le transforme en un PLTA *fusionné déterministe* $\mathcal{A}_d$ selon une partition Π, est basé sur celui des automates d'arbres usuels. La différence réside dans le fait que deux λ-transitions $\lambda_1 \to q_1$ et $\lambda_2 \to q_2$ sont fusionnées dans $\lambda_1 \sqcap \lambda_2 \to \{q_1, q_2\}$ quand λ_1 et λ_2 sont inclusent dans le même élément π de la partition Π (voir figure 5.5). Par conséquent, l'automate résultant de cet algorithme reconnait un langage plus grand : $\mathcal{L}(\mathcal{A}) \subseteq \mathcal{L}(\mathcal{A}_d)$. Cet algorithme produit la meilleure sur-approximation en terme d'inclusion de langage.

Algorithme de déterminisation :
Entrée : PLTA $\mathcal{A} = \langle \Pi, \mathcal{F}, \mathcal{Q}, \mathcal{Q}_f, \Delta \rangle$
Début
  $\mathcal{Q}_d := \emptyset$; $\Delta_d = \emptyset$;
  Pour tout $\pi \in \Pi$ **faire**
    $Trans(\pi) := \{\lambda \to q \in \Delta | \lambda \in \Lambda, \lambda \sqsubseteq \pi\}$;
    $s := \{q \in \mathcal{Q} | \lambda \to q \in Trans(\pi)\}$;
    $\mathcal{Q}_d := \mathcal{Q}_d \cup \{s\}$;

$\lambda_m := \bigsqcup\{\lambda | \lambda \to q \in Trans(\pi)\}$;
$\Delta_d := \Delta_d \cup \{\lambda_m \to s\}$;
Fin pour
Répéter
Soit $f \in \mathcal{F}_n$, $s_1, \ldots, s_n \in \mathcal{Q}_d$,
$s := \{q \in \mathcal{Q} | \exists q_1 \in s_1, \ldots, q_n \in s_n, f(q_1, \ldots, q_n) \to q \in \Delta\}$;
$\mathcal{Q}_d := \mathcal{Q}_d \cup \{s\}$;
$\Delta_d := \Delta_d \cup \{f(s_1, \ldots, s_n) \to s\}$;
Jusqu'à ce que qu'aucune règle ne puisse être encore ajoutée à Δ_d
$\mathcal{Q}_{df} := \{s \in \mathcal{Q}_d | s \cap \mathcal{Q}_f \neq \emptyset\}$
Sortie : Un PLTA *fusionné déterministe* $\mathcal{A}_d = \langle \Pi, \mathcal{F}, \mathcal{Q}_d, \mathcal{Q}_{df}, \Delta_d \rangle$
Fin

EXEMPLE *5.13*
Soit $\Delta = \{ \quad [-3,-1] \to q_1, [-5,-2] \to q_2,$
$[3,4] \to q_3, [-3,2] \to q_4,$
$f(q_1,q_2) \to q_5, f(q_3,q_4) \to q_6,$
$f(q_5,q_6) \to q_{f1}, f(q_5,q_6) \to q_{f2}\}$, et $\Pi = \{]-\infty, 0[, [0,0],]0, +\infty[\}$
Avec l'algorithme de déterminisation défini plus haut, nous obtenons l'ensemble de transitions déterministe suivant :
$\Delta_d = \{ \quad [-5,0[\to q_{1,2,4},]0,4] \to q_{3,4},$
$[0,0] \to q_4, f(q_{1,2,4}, q_{1,2,4}) \to q_5,$
$f(q_{3,4}, q_{3,4}) \to q_6, f(q_{3,4}, q_4) \to q_6,$
$f(q_{3,4}, q_{1,2,4}) \to q_6, f(q_5, q_6) \to q_{f1,f2}\}$.

◀

PROPOSITION 5.3.3 (Un PLTA déterminisé par l'algorithme est la meilleure sur-approximation)
Soit $\mathcal{A}_1$ un PLTA et $\mathcal{A}_2$ le PLTA obtenu grâce à l'algorithme de déterminisation. Alors $\mathcal{A}_2$ est la meilleure sur-approximation de $\mathcal{A}_1$ en tant que PLTA ***fusionné déterministe****.*

1. $\mathcal{L}(\mathcal{A}_1) \subseteq \mathcal{L}(\mathcal{A}_2)$
2. *Pour chaque PLTA* ***fusionné déterministe*** *$\mathcal{A}_3$ basé sur la même partition que $\mathcal{A}_1$,* $\mathcal{L}(\mathcal{A}_1) \subseteq \mathcal{L}(\mathcal{A}_3) \implies \mathcal{L}(\mathcal{A}_2) \subseteq \mathcal{L}(\mathcal{A}_3)$

Démonstration. (1) Cas de base : pour toute λ-transition $\lambda \to q$ de $\mathcal{A}_1$, soit $\pi \in \Pi$ tel que $\lambda \sqsubseteq \pi$. Alors $Trans(\pi) = \{\lambda \to q \in \Delta | \lambda \in \Lambda, \lambda \sqsubseteq \pi\}$. Il exisite donc une transition $\lambda' \to Q$ dans $\mathcal{A}_2$ telle que $\lambda' = \bigsqcup\{\lambda | \lambda \to q \in Trans(\pi)\}$ et $Q = \{q | \lambda \to q \in Trans(\pi)\}$, avec $q \in Q$.
Cas d'induction : pour toute non λ-transition $f(q_1, \ldots, q_n) \to q$ de $\mathcal{A}_1$, il existe la transition correspondante $f(Q_1, \ldots, Q_n) \to Q$ telle que $q \in Q$. Nous avons alors $q_1 \in Q_1, \ldots, q_n \in Q_n$ grâce à l'hypothèse d'induction.
Donc $\mathcal{L}(\mathcal{A}_1) \subseteq \mathcal{L}(\mathcal{A}_2)$.

(2) $\mathcal{A}_1 = \langle \Pi, \mathcal{F}, \mathcal{Q}_1, \mathcal{Q}_{f_1}, \Delta_1 \rangle$, $\mathcal{A}_2 = \langle \Pi, \mathcal{F}, \mathcal{Q}_2, \mathcal{Q}_{f_2}, \Delta_2 \rangle$ et $\mathcal{A}_3 = \langle \Pi, \mathcal{F}, \mathcal{Q}_3, \mathcal{Q}_{f_3}, \Delta_3 \rangle$.
Comme $\mathcal{L}(\mathcal{A}_1) \subseteq \mathcal{L}(\mathcal{A}_2)$ (1) et $\mathcal{L}(\mathcal{A}_1) \subseteq \mathcal{L}(\mathcal{A}_3)$, soient $\mathcal{R}_1 : \mathcal{Q}_1 \times \mathcal{Q}_2$ et $\mathcal{R}_2 : \mathcal{Q}_1 \times \mathcal{Q}_3$ deux relations de simulation définissant ces propriétés comme suit :

Soient $q_1 \in \mathcal{Q}_1$ et $q_2 \in \mathcal{Q}_2$, $(q_1, q_2) \in \mathcal{R}_1$ si et seulement si
- $\lambda_1 \to q_1 \in \Delta_1,\ \lambda_2 \to q_2 \in \Delta_2$ et $\lambda_1 \sqsubseteq \lambda_2$, avec $\lambda_1, \lambda_2 \in \Lambda$,
  ou $f(q_{i_1}, \ldots, q_{i_n}) \to q_1 \in \Delta_1,\ f(q'_{i_1}, \ldots, q'_{i_n}) \to q_2 \in \Delta_2$ et $\forall j \in [1, n],\ (q_{i_j}, q'_{i_j}) \in \mathcal{R}_1$, avec $f \in \mathcal{F}_n$
- $q_1 \in \mathcal{Q}_{f_1} \iff q_2 \in \mathcal{Q}_{f_2}$.

Soient $q_1 \in \mathcal{Q}_1$ et $q_3 \in \mathcal{Q}_3$, $(q_1, q_3) \in \mathcal{R}_2$ si et seulement si
- $\lambda_1 \to q_1 \in \Delta_1,\ \lambda_3 \to q_3 \in \Delta_3$ et $\lambda_1 \sqsubseteq \lambda_3$, avec $\lambda_1, \lambda_3 \in \Lambda$,
  ou $f(q_{i_1}, \ldots, q_{i_n}) \to q_1 \in \Delta_1,\ f(q'_{i_1}, \ldots, q'_{i_n}) \to q_3 \in \Delta_2$ et $\forall j \in [1, n],\ (q_{i_j}, q'_{i_j}) \in \mathcal{R}_2$, avec $f \in \mathcal{F}_n$
- $q_1 \in \mathcal{Q}_{f_1} \iff q_3 \in \mathcal{Q}_{f_3}$.

Soit $\mathcal{R} : \mathcal{Q}_2 \times \mathcal{Q}_3$ une relation de simulation telle que $(q_2, q_3) \in \mathcal{R}$ si et seulement si $\exists q_1 \in \mathcal{Q}_1.(q_1, q_2) \in \mathcal{R}_1 \wedge (q_1, q_3) \in \mathcal{R}_2$, avec $q_2 \in \mathcal{Q}_2$, $q_3 \in \mathcal{Q}_3$.

Soit $(q_2, q_3) \in \mathcal{R}$. Cela signifie que :
- $\lambda_1 \to q_1 \in \Delta_1,\ \lambda_2 \to q_2 \in \Delta_2,\ \lambda_3 \to q_3 \in \Delta_2$ et $\lambda_1 \sqsubseteq \lambda_2$ et $\lambda_1 \sqsubseteq \lambda_3$, avec $\lambda_1, \lambda_2, \lambda_3 \in \Lambda$ (a),
  ou $f(q_{i_1}, \ldots, q_{i_n}) \to q_1 \in \Delta_1,\ f(q'_{i_1}, \ldots, q'_{i_n}) \to q_2 \in \Delta_2,\ f(q''_{i_1}, \ldots, q''_{i_n}) \to q_3 \in \Delta_3$ et $\forall j \in [1, n],\ (q_{i_j}, q'_{i_j}) \in \mathcal{R}_1$ et $(q_{i_j}, q''_{i_j}) \in \mathcal{R}_2$, avec $f \in \mathcal{F}_n$ (b)
- $q_1 \in \mathcal{Q}_{f_1} \iff q_2 \in \mathcal{Q}_{f_2}$ et $q_1 \in \mathcal{Q}_{f_1} \iff q_3 \in \mathcal{Q}_{f_3}$ (c),

par définition de $\mathcal{R}_1$ et $\mathcal{R}_2$.

(a) Soit $\pi \in \Pi$ un élément de la partition tel que $\lambda_1 \sqsubseteq \pi$. Alors $Trans(\pi) = \{\lambda \to q \in \Delta | \lambda \in \Lambda, \lambda \sqsubseteq \pi\}$, *i.e.* l'ensemble de toutes les λ-transitions $\lambda \to q$ dans Δ_1 telles que $\lambda \sqsubseteq \pi$. Bien sur, $\lambda_1 \sqsubseteq Trans(\pi)$, car $\lambda_1 \sqsubseteq \pi$. Alors λ_2 est le lub de tous les $\lambda \in \Lambda$ tels que $\lambda \to q \in Trans(\pi)$, *i.e.* $\lambda_2 = \bigsqcup\{\lambda | \lambda \to q \in Trans(\pi)\}$, selon l'algorithme de déterminisation. Comme le lub $\sqcup$ de deux éléments d'un treillis est la meilleure et unique sur-approximation, cet algorithme de déterminisation retourne la meilleure sur-approximation.

Comme $\mathcal{A}_3$ est déterministe et contient $\mathcal{A}_1$, alors λ_3 doit contenir au moins tous les $\lambda \in \Lambda$ tels que $\lambda \to q \in \Delta_1$ et $\lambda \sqsubseteq \pi$, ou bien $\mathcal{A}_3$ n'est pas déterministe.

Donc $\lambda_3 \sqsupseteq \bigsqcup\{\lambda | \lambda \to q \in Trans(\pi)\}$, donc $\lambda_2 \sqsubseteq \lambda_3$.

(b) On peut immédiatement déduire que $\forall j \in [1, n],\ (q'_{i_j}, q''_{i_j}) \in \mathcal{R}$ par définition de $\mathcal{R}$.

(c) Donc $q_2 \in \mathcal{Q}_{f_2} \iff q_3 \in \mathcal{Q}_{f_3}$

Grâce à ces propriétés déduites de $\mathcal{R} : \mathcal{Q}_1 \times \mathcal{Q}_2$, on peut déduire que $\mathcal{L}(\mathcal{A}_2) \subseteq \mathcal{L}(\mathcal{A}_3)$.

□

Remarque. La minimisation d'un LTA s'effectue sur un LTA *déjà déterminisé* et n'est par conséquent pas définies pour tous les LTA. La minimisation calcule donc une sur-approximation et l'algorithme est le même que pour un automate d'arbres classiques (voir [Comon et al., 2007]), car les λ-transitions ne nécessitent pas de minimisation : la minimisation de ces transitions est déjà effectuée après déterminisation du LTA.

5.3.3 Raffinement de la partition

Dans la sous-section précédente, la partition Π était fixée. La précision de la sur-approximation apportée durant l'algorithme de déterminisation dépend de la finesse de

Π. Par exemple, si Π est de taille 1, toutes les λ-transitions seront fusionnées en une seule.

DÉFINITION 5.3.4 (Raffinement d'une partition)
Une partition Π_2 *raffine une partition* Π_1 *si :*

$$\forall \pi_2 \in \Pi_2,\ \exists \pi_1 \in \Pi_1 :\ \pi_2 \subseteq \pi_1$$

Soit $\mathcal{A}_1 = \langle \Pi_1, \mathcal{F}, \mathcal{Q}, \mathcal{Q}_f, \Delta_1 \rangle$ *un PLTA. Le PLTA* $\mathcal{A}_2 = \langle \Pi_2, \mathcal{F}, \mathcal{Q}, \mathcal{Q}_f, \Delta_2 \rangle$ *raffine* $\mathcal{A}_1$ *si :*

1. Π_2 *raffine* Π_1
2. *les transitions de* Δ_2 *sont obtenues par :* $\forall \lambda \to q \in \Delta_1,\ \forall \pi_2 \in \Pi_2,\ \lambda \sqcap \pi_2 \to q \in \Delta_2$

Raffiner un PLTA ne modifie pas immédiatement le langage reconnu, mais conduit à une sur-approximation plus précise lors de la déterminisation, comme illustré par l'exemple suivant.

EXEMPLE *5.14*
Soient Π et Δ de l'exemple 5.13 et une partition $\Pi_2 = \{]-\infty, -1[, [-1, 0[, [0, 0],]0, +\infty[\}$ qui raffine Π, l'ensemble des transitions Δ_2 du PLTA obtenu avec Π_2 est $\Delta_2 = \{[-3, -1[\to q_1, [-1, -1] \to q_1, [-5, -2] \to q_2, [3, 4] \to q_3, [-3, -1[\to q_4, [-1, 0[\to q_4, [0, 0] \to q_4,]0, 2] \to q_4, f(q_1, q_2) \to q_5, f(q_3, q_4) \to q_6, f(q_5, q_6) \to q_{f1}, f(q_5, q_6) \to q_{f2}\}$.

Après déterminisation, cet ensemble de transitions est obtenu avec Π_2 pour le PLTA déterministe correspondant : $\Delta_{2_d} = \{[-5, -1[\to q_{1,2,4}, [-1, 0[\to q_{1,4},]0, 4] \to q_{3,4}, [0, 0] \to q_4, f(q_{1,2,4}, q_{1,2,4}) \to q_5, f(q_{1,4}, q_{1,2,4}) \to q_5, f(q_{3,4}, q_{3,4}) \to q_6, f(q_{3,4}, q_4) \to q_6, f(q_{3,4}, q_{1,2,4}) \to q_6, f(q_{3,4}, q_{1,4}) \to q_6, f(q_5, q_6) \to q_{f1,f2}\}$.

◀

5.4 Un algorithme de complétion pour les LTA

Notre principal intérêt est de calculer l'ensemble des états accessibles d'un système à espace d'états infini. En général, cet ensemble n'est ni représentable, ni calculable (voir chapitres 1 et 2). Dans cette thèse, nous utilisons la méthode du Model-Checking régulier sur arbres (Regular Tree Model-Checking : RTMC) afin de pouvoir représenter des ensembles potentiellement infinis d'états. Plus précisément, dans ce chapitre, nous proposons de représenter les états (ou configurations) par des termes *interprétables* et les ensembles d'états par un LTA, défini à la section précédente. Nous rappelons, comme il est indiqué en section 5.3, que nous utilisons des LTA *non-déterministes*, et qu'*aucune* des étapes du calcul des états accessibles ne nécessite une déterminisation.

Ensuite, nous supposons que le comportement du système peut être représenté par un système de réécriture *conditionnel* (*CTRS* pour Conditional Term Rewriting System), *i.e.* dont chaque règle peut être équipée d'une conjonction de conditions utilisées pour restreindre l'applicabilité de la règle. Nos systèmes de réécriture conditionnels (*CTRS*), dont la définition étend celle de [Baader et Nipkow, 1998], peuvent réécrire les termes contenant des symboles interprétables appartenant au *domaine concret*. Définir les *CTRS* sur le domaine concret $\mathcal{C}$ plutôt que l'abstrait permet de les rendre indépendant du treillis utilisé. Ainsi, pour un même domaine concret, le système de réécriture reste le même quelque soit le treillis utilisé dans le LTA.

Commençons tout d'abord par la définition des prédicats permettant d'exprimer les conditions utilisées dans les *CTRS*.

DÉFINITION 5.4.1 (Prédicats)
Soit $\mathcal{P}$ l'ensemble des prédicats du domaine $\mathcal{C}$. Soit ρ un prédicat d'arité n de l'ensemble $\mathcal{P}$ tel que $\rho : \mathcal{C}^n \mapsto \{true, false\}$. Le domaine de ρ est étendu à $\mathcal{T}(\mathcal{F}, \mathcal{X})^n$ de la manière suivante :

$$\rho(t_1, \ldots, t_n) = \begin{cases} \rho(u_1, \ldots, u_n) \text{ si } \forall i = 1 \ldots n : t_i \in \mathcal{T}(\mathcal{F}_\bullet) \text{ et } u_i = eval(t_i) \\ faux \text{ si } \exists j = 1 \ldots n : t_j \notin \mathcal{T}(\mathcal{F}_\bullet) \end{cases}$$

On peut observer que les prédicats sont définis sur les termes interprétables du domaine concret. Si l'un des paramètres du prédicat ne peut être interprété, alors le prédicat retourne faux, afin d'empêcher l'application de la règle dans un tel cas.

EXEMPLE *5.15*
Les prédicats $x < 2$, $2 > 3$, $x \geq y + 6 \in \mathcal{P}$ sont évalués dans le domaine des entiers relatifs. Le prédicat $f(5)$ ne peut pas être interprété, il est donc évalué à faux. ◀

Nous pouvons maintenant définir formellement les *CTRS* utilisant des termes interprétables.

DÉFINITION 5.4.2 (Système de réécriture conditionnel (*CTRS*) sur $\mathcal{T}(\mathcal{F}_\circ \cup \mathcal{F}_\bullet, \mathcal{X})$)
Un CTRS $\mathcal{R}$ est un ensemble de règles de réécriture de la forme $l \to r \Leftarrow c_1 \wedge \ldots \wedge c_n$, avec $l \in \mathcal{T}(\mathcal{F}_\circ, \mathcal{X})$, $r \in \mathcal{T}(\mathcal{F}_\circ \cup \mathcal{F}_\bullet, \mathcal{X})$, $l \notin \mathcal{X}$, $\mathcal{V}ar(l) \supseteq \mathcal{V}ar(r)$ et $\forall i = 1 \ldots n : c_i = \rho_i(t_1, \ldots, t_m)$ avec ρ_i un prédicat d'arité m de l'ensemble $\mathcal{P}$ et $\forall j = 1 \ldots m : t_j \in \mathcal{T}(\mathcal{F}_\bullet, \mathcal{X}) \wedge \mathcal{V}ar(t_j) \subseteq \mathcal{V}ar(l)$.

EXEMPLE *5.16*
En utilisant des règles de réécritures conditionnelles, la fonction factorielle peut être définie de la manière suivante : $\begin{array}{lll} fact(x) \to & 1 & \Leftarrow x \geq 0 \wedge x \leq 1 \\ fact(x) \to & x * fact(x-1) & \Leftarrow x \geq 2 \end{array}$ ◀

Définissons maintenant la relation de réécriture des *CTRS* définis sur $\mathcal{T}(\mathcal{F}_\circ \cup \mathcal{F}_\bullet, \mathcal{X})$, utilisant la définition 2.1.8 des substitutions.

DÉFINITION 5.4.3 ($\to_\mathcal{R}$)
Soit $\mathcal{F} = \mathcal{F}_\circ \cup \mathcal{F}_\bullet$, le CTRS $\mathcal{R}$ et la fonction $eval$ produisent une relation de réécriture $\to_\mathcal{R}$ sur $\mathcal{T}(\mathcal{F})$ de la manière suivante : pour tout $s, t \in \mathcal{T}(\mathcal{F})$, on a $s \to_\mathcal{R} t$ s'il existe :

1. *une régle de réécriture $l \to r \Leftarrow c_1 \wedge \ldots \wedge c_n \in \mathcal{R}$,*
2. *une position $p \in \mathcal{P}os(s)$, et*
3. *une substitution $\sigma : \mathcal{X} \mapsto \mathcal{T}(\mathcal{F})$*

telles que $s|_p = l\sigma$, $t = eval(s[r\sigma]_p)$ et $\forall i = 1 \ldots n : c_i\sigma = true$.

La fermeture réflexive transitive de $\to_\mathcal{R}$ est notée $\to_\mathcal{R}^*$.

EXEMPLE *5.17* ($\to_\mathcal{R}$ et $\to_\mathcal{R}^*$)
Si l'on reprend les règles de réécriture de l'exemple 5.16, on a alors :
$fact(3) \to_\mathcal{R} eval(3 \times fact(3-1))$ car 3 vérifie la condition $x \geq 2$ (on utilise donc la

deuxième règle). Puis $eval(3 \times fact(3-1)) = 3 \times fact(2) \rightarrow_{\mathcal{R}} eval(3 \times 2 \times fact(2-1)$ car 2 vérifie $x \geq 2$. Puis enfin, on a $eval(3 \times 2 \times fact(2-1) = 6 \times fact(1) \rightarrow_{\mathcal{R}} eval(6 \times 1)$ car 1 vérifie la condition $x \geq 0 \wedge x \leq 1$ et la première règle est donc appliquée. Étant donné que $eval(6 \times 1) = 6$, on a alors $fact(3) \rightarrow^*_{\mathcal{R}} 6$. ◀

Soit $\mathcal{A}$ un LTA représentant l'ensemble des états initiaux, et $\mathcal{R}$ un *CTRS*. Notre objectif est de calculer un autre LTA représentant l'ensemble (ou une sur-approximation de l'ensemble) des états accessibles, décrit par : $\mathcal{R}^*(\mathcal{L}(\mathcal{A})) = \{t | \exists t_0 \in \mathcal{L}(\mathcal{A}), t_0 \rightarrow^*_{\mathcal{R}} t\}$.

Dans ce chapitre, nous adaptons l'algorithme de [Genet et Rusu, 2010] expliqué en section 2.4, qui permet de calculer un automate d'arbre $\mathcal{A}^k_{\mathcal{R}}$ tel que $\mathcal{L}(\mathcal{A}^k_{\mathcal{R}}) \supseteq \mathcal{R}^*(\mathcal{L}(\mathcal{A}))$ pour un système de réécriture *linéaire gauche* $\mathcal{R}$ (voir définition 2.1.16). Nous rappelons succinctement que cet algorithme calcule successivement la séquence d'automates $\mathcal{A}^0_{\mathcal{R}}, \mathcal{A}^1_{\mathcal{R}}, \mathcal{A}^2_{\mathcal{R}}, ...$ qui représente les applications successives de $\mathcal{R}$. Calculer $\mathcal{A}^{i+1}_{\mathcal{R}}$ à partir de $\mathcal{A}^i_{\mathcal{R}}$ se fait en *une étape de complétion*.

Rappelons également que généralement, cette séquence d'automates ne converge pas en un temps fini. Pour accélérer la convergence, il existe diverses méthodes et opérations d'abstractions décrites en section 3.3. La méthode d'abstraction utilisée ici découle de l'*abstraction équationnelle*, et sera décrite dans la sous-section 5.4.3. Nous commencerons tout d'abord par décrire le calcul de l'automate successeur $\mathcal{A}^{i+1}_{\mathcal{R}}$. Puis, nous montrerons que, pour conserver la correction, notre procédure doit être combinée avec une évaluation du LTA courant. Cette évaluation, décrite dans la sous-section 5.4.2, peut ajouter de nouveaux termes au langage de l'automate évalué, et ainsi ajouter un nombre potentiellement infini de transitions. Ce comportement infini sera capturé par une opération de *widening* utilisée généralement dans le domaine de l'interprétation abstraite. Toutes ces constructions seront illustrées étapes par étapes par un exemple fil rouge.

5.4.1 Calcul du LTA successeur

Rappelons, comme expliqué en section 2.4, que dans la complétion classique, l'automate successeur $\mathcal{A}^{i+1}_{\mathcal{R}}$ est construit à partir de $\mathcal{A}^i_{\mathcal{R}}$ en appliquant une *étape de complétion* consistant à trouver des paires critiques. Soit une substitution $\sigma : \mathcal{X} \mapsto \mathcal{Q}$ et une règle conditionnelle $l \rightarrow r \Leftarrow c_1 \wedge \ldots \wedge c_n \in \mathcal{R}$, une paire critique est une paire $(r\sigma, q)$ avec $q \in \mathcal{Q}$, telle que $l\sigma \rightarrow^*_{\mathcal{A}^i_{\mathcal{R}}} q$ et $r\sigma \nrightarrow^*_{\mathcal{A}^i_{\mathcal{R}}} q$.

Trouver l'ensemble des paires critiques d'une règle de réécriture requiert la détection de toutes les substitutions $\sigma : \mathcal{X} \mapsto \mathcal{Q}$ telles que $l\sigma \rightarrow^* q$, avec q un état de l'automate. Pour cela nous utilisons l'algorithme standard de *filtrage* détaillé en section 2.4.1.

Cet algorithme, $Matching(l, \mathcal{A}, q)$, trouve la correspondance entre un terme *linéaire* l et un état q dans l'automate $\mathcal{A}$. La solution retournée par l'algorithme de filtrage est un ensemble de substitutions $\{\sigma_1, \ldots, \sigma_n\}$ telles que $l\sigma_i \rightarrow^*_{\mathcal{A}} q$ avec $i \in [1, n]$.

Dans le cas classique, chaque paire critique $(r\sigma, q)$ entraine l'ajout d'une transition $r\sigma \rightarrow q$ (ou d'un ensemble de transitions si celle-ci n'est pas normalisée) dans l'automate courant. Cependant, cette étape nécessite d'être modifiée dans le cas de la complétion pour les LTA. En effet, comme les *CTRS* contiennent des prédicats conditionnant l'applicabilité de la règle, nous devons étendre cet algorithme pour garantir que chaque substitution σ_i, trouvée par l'algorithme de filtrage pour une règle conditionnelle $l \rightarrow r \Leftarrow c_1 \wedge \ldots \wedge c_n$ donnée, satisfait $c_1 \wedge \ldots \wedge c_n$.

EXEMPLE *5.18*
Prenons l'ensemble $\mathbb{Z}$ en tant que domaine concret, et l'ensemble des intervalles sur $\mathbb{Z} \cup \{-\infty, +\infty\}$ en tant que treillis abstrait. Soient $\mathcal{R} = \{f(x) \to cons(x, f(x+1)) \Leftarrow x \geq 1\}$ le *CTRS*, et $\mathcal{A}_0$ le LTA représentant l'ensemble des états initiaux, possédant l'ensemble de transitions suivant :
$\Delta_0 = \{[0,2] \to q_1, f(q_1) \to q_2\}$.

Pour construire $\mathcal{A}^1_{\mathcal{R}}$ à partir de $\mathcal{A}_0$, toutes les substitutions possibles doivent être calculées. Ici, l'algorithme de filtrage retourne la substitution $\{x \mapsto q_1\}$. Or $[0,2] \to q_1 \in \Delta$ et $[0,2]$ ne satisfait pas la contrainte $x \geq 1$. La substitution doit donc être restreinte à $\{x \mapsto [1,2]\}$. ◀

La restriction des substitutions est faite par un solveur $Solve$ définit sur le domaine abstrait choisi. Un tel *solveur* prend en entrée les substitutions calculées par l'algorithme de filtrage classique, les λ-transitions de l'automate ainsi que toutes les conditions de la règles. $Solve(\sigma, \mathcal{A}, c_1 \wedge \ldots \wedge c_n)$ retourne :

- soit un ensemble de substitutions σ' correspondant à une restriction de σ satisfaisant $c_1 \wedge \ldots \wedge c_n$,
- soit $\emptyset$ si une telle restriction n'existe pas.

Sur l'exemple précédent, $Solve(\{x \mapsto q_1\}, \mathcal{A}, x \geq 1) = \{\{x \mapsto [1,2]\}\}$. Certains solveurs existent pour des domaines abstraits assez variés (voir [Pichardie, 2005]). Dans notre contexte, le solveur doit satisfaire la propriété suivante :

Propriété 1 (Correction du solveur). Soient $\sigma = \{x_1 \mapsto q_1, \ldots, x_k \mapsto q_k\}$ une substitution et $c = c_1 \wedge \cdots \wedge c_n$ une conjonction de contraintes. Soit $\sigma/c = \{x_i \mapsto q_i \mid \exists 1 \leq j \leq n, x_i \in \mathcal{V}ar(c_j)\}$ l'ensemble des substitutions dont la variable figure dans l'ensemble de contraintes. Définissons également $S_c = \{i \mid \exists 1 \leq j \leq n, x_i \in \mathcal{V}ar(c_j)\}$.

Pour chaque tuple $\langle \lambda_i | i \in S_c \rangle$ tel que $\lambda_i \to^*_{\mathcal{A}} q_i$, $Solve_\Lambda(\sigma/c, \langle \lambda_i | i \in S_c \rangle, c)$ est une substitution σ' telle que

(1) si $i \notin S_c$, $\sigma'(x_i) = q_i$, et

(2) si $i \in S_c$, $\sigma'(x_i) = \lambda'_i$.

De plus, si un tuple de valeurs abstraites $\langle \lambda''_i | i \in S_c \rangle$, satisfait

(a) $\forall i \in S_c$, $\lambda''_i \sqsubseteq \lambda_i$, et

(b) $\forall 1 \leq j \leq n$, la substitution $\sigma''/c = \{x_i \mapsto \lambda''_i\}$ satisfait c_j,

alors $\forall i \in S_c$, $\lambda''_i \sqsubseteq \lambda'_i$, *i.e.*, le tuple solution $\langle \lambda'_i | i \in S_c \rangle$ est une sur-approximation de la solution.

De telles propriétés sont généralement respectées par les implémentations des domaines abstraits usuels.

En utilisant la proposition 1, la fonction $Solve(\sigma, \mathcal{A}, c_1 \wedge \cdots \wedge c_n)$ est définie comme suit :

$$Solve(\sigma, \mathcal{A}, c_1 \wedge \cdots \wedge c_n) = \bigcup_{\lambda_1 \to^*_{\mathcal{A}} q_1, \ldots, \lambda_k \to^*_{\mathcal{A}} q_k} Solve_\Lambda(\sigma/c, \langle \lambda_i | i \in S_c \rangle, c)$$

Le théorème suivant garantit que $Solve(\sigma, \mathcal{A}, c_1 \wedge \cdots \wedge c_n)$ est une sur-approximation des solutions de l'ensemble de contraintes.

THÉORÈME 5.4.4
$Solve(\sigma, \mathcal{A}, c_1 \wedge \cdots \wedge c_n)$ est une sur-approximation des solutions de l'ensemble de contraintes.

Démonstration. Pour chaque tuple $\langle \lambda_i | i \in S_c \rangle$ tel que $\lambda_i \rightarrow^*_{\mathcal{A}} q_i$, alors on a par la proposition 1 : $Solve_\Lambda(\sigma/c, \langle \lambda_i | i \in S_c \rangle, c)$ est une substitution σ' telle que si $i \in S_c$, $\sigma'(x_i) = \lambda'_i$. Soit $\langle \lambda''_i | i \in S_c \rangle$ un tuple tel que $\forall 1 \leq j \leq n$, la substitution $\sigma''/c = \{x_i \mapsto \lambda''_i\}$ satisfait c_j. Grâce à la proposition 1, on a : $\forall i \in S_c$, $\lambda''_i \sqsubseteq \lambda'_i$. Comme tout $i \in S_c$, λ'_i est retourné par le solveur, on peut alors déduire que l'ensemble des substitutions retourné par le solveur est une sur-approximation des solutions de l'ensemble de contraintes. □

Définir un solveur qui satisfait cette propriété peut s'avérer complexe, selon le domaine abstrait Λ et le type de contraintes de c utilisé. Cependant, nous allons voir qu'une solution facile peut déjà être obtenue si c est une conjonction de contraintes linéaires et Λ le treillis des intervalles. L'algorithme permettant de calculer $Solve_\Lambda(\sigma, \langle \lambda_1, \ldots, \lambda_k \rangle, c_1 \wedge \cdots \wedge c_n)$ dans ce cas particulier est le suivant :

1. P_1 est le polyèdre convexe défini par les contraintes $c_1 \wedge \cdots \wedge c_n$,
2. P_2 est la boite définie par les contraintes $x_1 \in \lambda_1, \ldots x_k \in \lambda_k$,
3. si $P_1 \sqcap P_2 \neq \bot$, alors $P_1 \sqcap P_2$ est projeté sur chaque dimension (*i.e.* sur chaque variable x_k) pour obtenir k nouveaux intervalles. Sinon, $Solve_\Lambda(\sigma, \langle \lambda_1, \ldots, \lambda_k \rangle, c_1 \wedge \cdots \wedge c_n) = \emptyset$.

On peut noter que cet algorithme fonctionne également pour d'autres treillis abstraits comme les octogones ou les polyhèdres convexes, mais l'approximation peut dans ce cas se révéler assez grossière. On peut cependant définir un solveur de manière plus fine grâce à la programmation linéaire.

Définissons maintenant formellement l'ensemble de substitutions retourné, après restriction par le solveur. En d'autres mots, l'ensemble de substitutions σ' réunissant les conditions nécessaires pour que $r\sigma' \rightarrow q$ soit ajouté dans l'automate courant.

DÉFINITION 5.4.5 (Substitutions du filtrage respectant les conditions d'une règle conditionnelle)
Soient $\mathcal{A}$ un automate d'arbres, $rl = l \rightarrow r \Leftarrow c_1 \wedge \ldots \wedge c_n$ une règle de réécriture et q un état de $\mathcal{A}$. L'ensemble de toutes les substitutions possibles pour la règle rl est défini par :

$$\begin{aligned} \Omega(\mathcal{A}, rl, q) = \{\ \sigma' \ \mid \ & \sigma \in Matching(l, \mathcal{A}, q) \\ & \wedge\ \sigma' \in Solve(\sigma, \mathcal{A}, c_1 \wedge \ldots \wedge c_n) \\ & \wedge\ \nexists \sigma'' : r\sigma' \sqsubseteq r\sigma'' \text{ et } r\sigma'' \rightarrow_{\mathcal{A}}{}^* q, \text{ si } r\sigma', r\sigma'' \in \Lambda\ (*)\ \}. \end{aligned}$$

$(*)$ *en d'autres mots, nul besoin d'ajouter une λ-transition $\lambda \rightarrow q$ dans $\mathcal{A}$ s'il existe déjà dans l'automate une transition qui inclut la valeur reconnue, i.e. si $\lambda' \rightarrow_{\mathcal{A}} q$ et $\lambda \sqsubseteq \lambda'$.*

Une fois l'ensemble des solutions restreintes σ' obtenu, il faut ajouter les règles de la forme $r\sigma' \rightarrow^* q$ dans l'automate. Cependant, ces transitions ne sont pas forcément des transitions *normalisées* (voir définition 2.2.2) de la forme $f(q_1, \ldots, q_n) \rightarrow q$ (transition close) ou de la forme $\lambda \rightarrow q$ (λ-transition). Dans ce cas, $r\sigma' \rightarrow^* q$ doit tout d'abord être normalisé afin d'être ajouté au LTA.

EXEMPLE *5.19*
Si l'on reprends l'exemple 5.18, on constate que la transition $cons([1,2], f([1,2]+1)) \rightarrow q'_2$ que l'on doit ajouter, n'est pas normalisée : entre autre, 1 doit être abstrait par rapport au treillis utilisé (1 sera ici abstrait en $[1,1]$) et $f([1,2])$ doit être remplacé par un état reconnaissant ce terme. Une λ-transition $[1,2] \rightarrow q_{new}$ devra donc être ajoutée. ◀

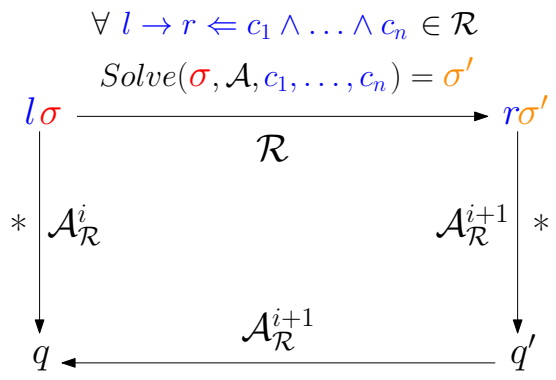


FIGURE 5.6 — Une étape de complétion dans le cas LTA et *CTRS*, utilisant un solveur.

Une telle normalisation, adaptée au LTA, est définie dans l'algorithme de normalisation suivant.

DÉFINITION 5.4.6 (Normalisation)
Soient $s \in \mathcal{T}(\mathcal{F} \cup \mathcal{Q})$, $q \in \mathcal{Q}$, $\mathcal{A} = \langle \mathcal{F}, \mathcal{Q}, \mathcal{Q}_f, \Delta \rangle$ un LTA, avec $\mathcal{F}_\bullet$ l'ensemble des symboles interprétables concrets utilisés dans le CTRS, $\mathcal{F}^\#_\bullet$ l'ensemble des symboles abstraits utilisés dans $\mathcal{A}$, $\mathcal{F} = \mathcal{F}^\#_\bullet \cup \mathcal{F}_\circ$, et $\alpha : \mathcal{F}^0_\bullet \to \mathcal{F}^{\#^0}_\bullet$ la fonction d'abstraction, associant les constantes concrètes aux éléments de Λ. Un nouvel état est un état de $\mathcal{Q}$ n'apparaissant pas dans Δ. $Norm(s \to q)$ retourne l'ensemble des transitions normalisées déduites de s. $Norm(s \to q)$ est défini inductivement par :

1. *si $s \in \mathcal{F}^0_\bullet$ (i.e., dans le domaine concret utilisé par le CTRS), alors $Norm(s \to q) = \{\alpha(s) \to q\}$.*
2. *si $s \in \mathcal{F}^0_\circ \cup \mathcal{F}^{0^\#}_\bullet$ alors $Norm(s \to q) = \{s \to q\}$,*
3. *si $s = f(t_1, \ldots, t_n)$, avec $f \in \mathcal{F}^n_\circ \cup \mathcal{F}^n_\bullet$, alors $Norm(s \to q) = \{f(q'_1, \ldots, q'_n) \to q\} \cup Norm(t_1 \to q'_1) \cup \ldots \cup Norm(t_n \to q'_n)$, où, pour $i = [1, n]$, q'_i est :*
   - *soit la partie droite d'une transition de Δ telle que $t_i \to^*_\Delta q'_i$*
   - *soit un nouvel état.*

Il est intéressant d'observer que cet algorithme termine toujours.

EXEMPLE *5.20*
Continuons notre exemple fil rouge. Soient $q_{[1,1]}, q_3, q_4, q_5$ de nouveaux états nécessaires à la normalisation, alors $Norm(cons([1,2], f([1,2]+1)) \to q'_2) = \{[1,2] \to q_3, [1,1] \to q_{[1,1]}, q_3 + q_{[1,1]} \to q_4, f(q_4) \to q_5, cons(q_3, q_5) \to q'_2\}$

Après calcul du nouvel ensemble de transitions après une étape de complétion, on obtient :

$$\begin{array}{lll} \Delta_1 & = \Delta_0 \cup & Norm(cons([1,2], f([1,2]+1)) \to q'_2) \cup \{q'_2 \to q_2\} \\ & = \Delta_0 \cup \{ & [1,2] \to q_3, [1,1] \to q_{[1,1]}, \\ & & q_3 + q_{[1,1]} \to q_4, f(q_4) \to q_5, \\ & & cons(q_3, q_5) \to q'_2, q'_2 \to q_2 \ \}, \end{array}$$

◀

Pour conclure cette sous-section, voici la caractérisation formelle d'une étape de complétion (voir aussi Fig.5.6).

DÉFINITION 5.4.7 (Une étape de complétion : calcul de l'automate successeur $\mathcal{C}_{\mathcal{R}}(\mathcal{A})$)
Soient $\mathcal{A} = \langle \mathcal{F}, \mathcal{Q}, \mathcal{Q}_f, \Delta \rangle$ un LTA, et $\mathcal{R}$ un CTR linéaire gauche. On note $\mathcal{C}_{\mathcal{R}}(\mathcal{A})$ le LTA obtenu après une étape de complétion. $\mathcal{C}_{\mathcal{R}}(\mathcal{A}) = \langle \mathcal{F}, \mathcal{Q}', \mathcal{Q}_f, \Delta' \rangle$ où :

$$\Delta' = \Delta \cup \bigcup_{l \to r \in \mathcal{R},\, q \in \mathcal{Q},\, \sigma \in \Omega(\mathcal{A}, l \to r, q)} Norm(r\sigma \to q') \cup \{q' \to q\}$$

avec $\Omega(\mathcal{A}, l \to r, q)$ l'ensemble de toutes les substitutions possibles définies dans la définition 5.4.5, $q' \notin \mathcal{Q}$ un nouvel état et $\mathcal{Q}'$ contenant tous les états de Δ'.

5.4.2 Évaluation d'un LTA

Chaque ensemble d'états contenant le terme interprétable $1 + 2$ devrait également contenir le terme 3. Bien que cette propriété canonique puisse être vérifiée de manière évidente sur l'ensemble des états initiaux, elle peut éventuellement être invalidée lors d'une étape de complétion ou en fusionnant des états (lors de l'étape d'abstraction, voir sous-section suivante).

EXEMPLE *5.21*
La première étape de complétion décrite dans l'exemple 5.19 permet d'ajouter la transition $q_3 + q_{[1,1]} \to q_4$. Comme on a $[1, 2] \to q_3$ et $[1, 1] \to q_{[1,1]}$, le langage reconnu par q_4 devrait aussi contenir le terme $[2, 3]$. ◀

L'objectif de la fonction $propag$ décrite ci-dessous est d'évaluer le LTA afin d'ajouter les transitions nécessaires pour compléter le langage, comme ajouter la transition $[2, 3] \to q_4$ pour l'exemple ci-dessus.

DÉFINITION 5.4.8 ($propag$)
Soit Δ l'ensemble de transitions d'un LTA. Soit $f(q_1, \ldots, q_n) \to q \in \Delta$, avec $f \in \mathcal{F}^n_\bullet$ un symbole interprétable et $q, q_1, \ldots, q_n \in \mathcal{Q}$. S'il existe $\lambda_1, \ldots, \lambda_n \in \Lambda$ tels que $\lambda_1 \to^*_\Delta q_1, \ldots, \lambda_n \to^*_\Delta q_n$, alors une étape d'évaluation de $f(q_1, \ldots, q_n) \to q$ est définie par :

$$propag(\Delta, f(q_1, \ldots, q_n) \to q) = \begin{cases} \Delta \text{ si } \exists \lambda \to q \in \Delta \wedge eval(f(\lambda_1, \ldots, \lambda_n)) \sqsubseteq \lambda \\ \Delta \cup \{eval(f(\lambda_1, \ldots, \lambda_n)) \to q\}, sinon. \end{cases}$$

Une étape d'évaluation pour Δ est définie par :

$$propag(\Delta) = \bigcup_{\forall f(q_1, \ldots, q_n) \to q \in \Delta \ t.q. \ f \in \mathcal{F}^n_\bullet} propag(\Delta, f(q_1, \ldots, q_n) \to q)$$

Comme la fonction $propag$ ajoute de nouvelles transitions, elle doit être appliquée jusqu'à ce qu'un point-fixe soit trouvé. En utilisant $propag$, nous pouvons donc étendre la fonction $eval$ aux ensembles de transitions et aux LTA de la manière suivante.

DÉFINITION 5.4.9 ($eval$ appliquée sur les transitions des LTA)
Soit $\mu X.f(X)$ le dernier point-fixe obtenu par itération d'une fonction générique f. Alors on a :
- $eval(\Delta) = \mu X.propag(X) \cup \Delta$ *et*
- $eval(\langle \mathcal{F}, \mathcal{Q}, \mathcal{Q}_f, \Delta \rangle) = \langle \mathcal{F}, \mathcal{Q}, \mathcal{Q}_f, eval(\Delta) \rangle$

EXEMPLE *5.22*
Dans notre exemple fil rouge, $eval(\Delta_1) = \Delta_1 \cup \{[2,3] \to q_4\}$. ◀

THÉORÈME 5.4.10
Pour tout LTA $\mathcal{A}$, $\mathcal{L}(\mathcal{A}) \subseteq \mathcal{L}(eval(\mathcal{A}))$

Démonstration. Par définition de $propag$ (définition 5.4.8), on a $propag(\Delta) = \Delta$ si $\exists \lambda \to q \in \Delta \wedge eval(\lambda_1 \bullet \ldots \bullet \lambda_k) \sqsubseteq \lambda$ ou $propag(\Delta) = \Delta \cup \{eval(\lambda_1 \bullet \ldots \bullet \lambda_k) \to q\}$. Dans tous les cas, $\Delta \subseteq propag(\Delta)$.

Par définition de $eval$ (définition 5.4.9), $eval(\Delta) = \mu X.propag(X) \cup \Delta$. Comme $\Delta \subseteq propag(\Delta)$, on a $\Delta \subseteq eval(\Delta)$. Alors on peut déduire que $\mathcal{L}(\mathcal{A}) \subseteq \mathcal{L}(eval(\mathcal{A}))$. □

5.4.3 Abstraction équationnelle

Comme nous l'avons déjà expliqué, la complétion peut ne pas converger. En effet, si l'on effectue une autre étape de complétion de notre exemple fil rouge, alors la règle de réécriture sera appliquée avec une nouvelle substitution $\{x \mapsto q_4\}$.

EXEMPLE *5.23*
Les transitions $Norm(cons(q_4, f(q_4+1))) \to q'_5$ et $q'_5 \to q_5$ vont donc être ajoutées à $eval(\Delta_1)$ pour construire $\mathcal{A}^2_{\mathcal{R}}$ à partir de $\mathcal{A}^1_{\mathcal{R}}$. On a alors :

$$\Delta_2 = eval(\Delta_1) \cup \{ \begin{array}{l} q_4 + q_{[1,1]} \to q_6, f(q_6) \to q_7, \\ cons(q_4, q_7) \to q'_5, q'_5 \to q_5 \ \}. \end{array}$$

Évaluons ce nouvel ensemble de transitions. On obtient $eval(\Delta_2) = \Delta_2 \cup \{[3,4] \to q_6\}$. Ce processus va continuer indéfiniment, car il va calculer le terme infini :

$$cons([1,2], cons([2,3], cons([3,4], \ldots))).$$

◀

La terminaison de la complétion peut être améliorée grâce à un ensemble E d'*équations d'approximation* comme dans [Meseguer et al., 2003, Genet et Rusu, 2010] (voir sections 2.4.2 et 3.3.6). Rappelons que selon l'objectif, l'ensemble E peut être défini manuellement [Meseguer et al., 2003], ou manuellement puis raffiné automatiquement [Boichut et al., 2012] (voir section 3.3.6), ou encore, généré automatiquement après analyse statique du système de réécriture (e.g. [Boichut et al., 2008b]).

Après analyse de l'exemple fil rouge, on constate que le comportement infini est provoqué par les transitions de la forme $q_i + q_{[1,1]} \to q_j$. Malheureusement, les équations classiques, définies sur $\mathcal{T}(\mathcal{F}_\circ, \mathcal{X})$, ne fonctionnent pas pour les termes possédant des symboles interprétables, et ne sont donc dans ce cas pas assez puissantes pour garantir la terminaison.

Une équation fonctionnant avec des symboles interprétables, *i.e.* une équation telle que $x = x + 1$ est nécessaire pour garantir la terminaison. Nous avons alors défini un nouveau type d'équations définies sur $\mathcal{T}(\mathcal{F}, \mathcal{X})$, pouvant contenir des conditions d'applicabilité de l'équation. De telles équations sont de la forme $u = v \Leftarrow c_1 \wedge \ldots \wedge c_n$, avec $u, v \in \mathcal{T}(\mathcal{F}_\circ \cup \mathcal{F}_\bullet, \mathcal{X})$. Le fonctionnement de l'abstraction grâce à ce nouveau type d'équation est résumé en figure 5.7.

Soit $\sigma : \mathcal{X} \mapsto \mathcal{Q}$ une substitution telle que $u\sigma \to_{\mathcal{A}^{i+1}_{\mathcal{R}}} q$, $v\sigma \to_{\mathcal{A}^{i+1}_{\mathcal{R}}} q'$ et $q \neq q'$. Une sur- approximation de $\mathcal{A}^{i+1}_{\mathcal{R}}$ (notée $\mathcal{A}^{i+1}_{\mathcal{R},E}$) peut être obtenue en fusionnant les états q et

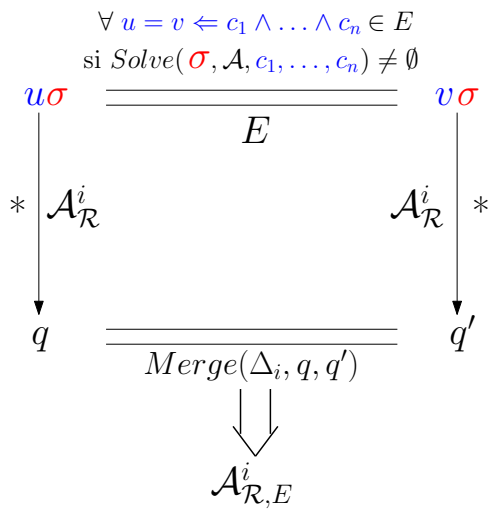


FIGURE 5.7 — Abstraction équationnelle pour LTA.

q'. Cette opération de fusion, dont le résultat est noté $Merge(\Delta, q, q')$ dans ce chapitre, consiste à remplacer chaque occurrence de q' par q dans $\mathcal{A}^{i+1}_{\mathcal{R}}$ (voir définition 2.4.8). Afin de trouver l'ensemble de toutes les substitutions vérifiant $u\sigma \rightarrow_{\mathcal{A}^{i+1}_{\mathcal{R}}} q$, on constate que l'on peut presque utiliser le même algorithme de *filtrage* que pour la complétion.

La première différence se situe au niveau du filtrage des termes $t \in \mathcal{T}(\mathcal{F}_\circ \cup \mathcal{F}_\bullet, \mathcal{X})$ possédant des symboles du domaine *concret*, qu'il faut faire correspondre à des termes de $\mathcal{T}(\mathcal{F}_\circ \cup \mathcal{F}^\#_\bullet, \mathcal{X})$ possédant des symboles treillis *abstrait*, reconnus par le LTA $\mathcal{A}$. Soit $\alpha : \mathcal{F}_\bullet \mapsto \mathcal{F}^\#_\bullet$ la fonction d'abstraction. Celle-ci va permettre de faire correspondre un symbole du domaine concret (par exemple, le chiffre 1 pour l'ensemble des entiers) à sa correspondance abstraite via la fonction α (ici, l'intervalle $[1, 1]$ si le treillis des intervalles est utilisé). Nous avons pour cela besoin d'ajouter une règle à l'algorithme de filtrage :

$$\text{(Constante du domaine concret)}\ \frac{d \trianglelefteq q}{(\exists\ \alpha(d) \rightarrow q \in \Delta) \vee \bot}, avec\ d \in \mathcal{F}^0_\bullet$$

La deuxième différence, c'est que contrairement au cas de la complétion, toutes les substitutions σ obtenues par l'algorithme de filtrage n'ont pas besoin d'être restreintes afin de respecter les contraintes de l'équation, il suffit seulement de vérifier que ces contraintes sont satisfiables, *i.e.* que $Solve(\sigma, \mathcal{A}, c_1 \wedge \cdots \wedge c_n) \neq \emptyset$ (voir figure 5.7).

EXEMPLE *5.24*
L'ensemble d'équations $E = \{x = x + 1 \Leftarrow x > 2\}$ peut être utilisé pour l'exemple 5.23. Il y a deux substitutions possibles : $\sigma_1 = \{x \mapsto q_3\}$ et $\sigma_2 = \{x \mapsto q_4\}$. La substitution σ_1 vient de la transition $q_3 + q_{[1,1]} \rightarrow q_4$. Cependant, comme $[1, 2] \rightarrow q_3$, on a $Solve(\{x \mapsto q_3\}, \mathcal{A}_2, x > 2) = \emptyset$, et σ_1 ne satisfait donc pas la condition. La substitution σ_2, qui vient de la transition $q_4 + q_{[1,1]} \rightarrow q_6$, satisfait la condition car $[2, 3] \rightarrow q_4$ et $Solve(\{x \mapsto q_4\}, \mathcal{A}_2, x > 2) = \{x \mapsto [3, 3]\} \neq \emptyset$. Par conséquent, l'équation est appliquée pour σ_2 et entraine la fusion des états q_4 et q_6. ◀

Remarque. Les équations classiques utilisées dans la complétion usuelle, seront également utilisées ici pour capturer les comportements infinis induits par les termes non-

interprétables de $\mathcal{T}(\mathcal{F}_\circ)$.

THÉORÈME 5.4.11
Soient $\mathcal{A} = \langle \mathcal{F}, \mathcal{Q}, \mathcal{Q}_f, \Delta \rangle$ un LTA, E un ensemble d'équations et $\sigma : \mathcal{X} \mapsto \mathcal{Q}$ une substitution. Deux états q_1 et q_2 de $\mathcal{A}$ sont dits équivalents par E s'il existe une équation $u = v$ de E telle que $u\sigma \to^*_\Delta q_1$ et $v\sigma \to^*_\Delta q_2$. On note $\rightsquigarrow^!_E$ la transformation de $\mathcal{A}$ après fusion d'états équivalents par E. Le langage du LTA résultat $\mathcal{A}'$ tel que $\mathcal{A} \rightsquigarrow^!_E \mathcal{A}'$, est une surapproximation du langage de $\mathcal{A}$, i.e. $\mathcal{L}(\mathcal{A}) \subseteq \mathcal{L}(\mathcal{A}')$. De plus, le calcul de $\mathcal{A}'$ termine (i.e. la transformation $\rightsquigarrow^!_E$ est convergente).

Démonstration. Soient $\mathcal{A}$ et $\mathcal{A}'$ deux LTA et E un ensemble d'équations, tels que $\mathcal{A} \rightsquigarrow^!_E \mathcal{A}'$. L'ensemble des transitions of $\mathcal{A}'$ est le même que celui de $\mathcal{A}$, à la différence près qu'il possède des états fusionnés selon les classes d'équivalences déterminées par E. Pour tout $t \in \mathcal{T}(\mathcal{F}, \mathcal{X})$, pour tout état q de $\mathcal{A}$, soit $Q = \{q_1, \ldots, q, \ldots, q_n\}$ l'ensemble des états rendus équivalents par E (*i.e.*, l'ensemble des états qui seront fusionnés après application de E).

Alors, comme $t \in \mathcal{L}(\mathcal{A}, q)$ implique que $t \to^*_\mathcal{A} q$, il reste à prouver que $t \to^*_\mathcal{A} q \implies t \to^*_{\mathcal{A}'} Q$. Procédons par récurrence.

Cas de base : Si $t = a$ est une constante, alors étant donné que $q \in Q$, $t \to^*_\mathcal{A} q \implies t \to^*_{\mathcal{A}'} Q$.

Induction : Soit $t = f(t_1, \ldots, t_n)$ avec $t_1, \ldots, t_n \in \mathcal{T}(\mathcal{F}, \mathcal{X})$ et $f \in \mathcal{F}^n$. Si $f(t_1, \ldots, t_n) \to^*_\mathcal{A} q$, alors cela signifie qu'il existe $q_1, \ldots, q_n$ des états de $\mathcal{A}$ tels que $t_1 \to^*_\mathcal{A} q_1, \ldots, t_n \to^*_\mathcal{A} q_n$ et $f(q_1, \ldots, q_n) \to q$ est une transition de $\mathcal{A}$ (1). Soient $Q_1, \ldots, Q_n$ correspondant respectivement aux classes d'équivalences des états $q_1, \ldots, q_n$, alors par hypothèse d'induction, on a $t_1 \to^*_{\mathcal{A}'} Q_1, \ldots, t_n \to^*_{\mathcal{A}'} Q_n$ (2). De (1) et (2) on peut alors déduire que $f(Q_1, \ldots, Q_n) \to Q$ est une transition de $\mathcal{A}'$, donc $t = f(t_1, \ldots, t_n) \to^*_{\mathcal{A}'} Q$. On a alors $t \in \mathcal{L}(\mathcal{A}', Q)$.

Chaque étape de la modification $\mathcal{A} \rightsquigarrow^!_E \mathcal{A}'$ remplace chaque occurrence d'un état q' par un état q dans l'ensemble de transitions de $\mathcal{A}$ afin d'obtenir $\mathcal{A}'$. Par conséquent, chaque étape de cette modification fait strictement décroitre le nombre d'états de l'automate, qui ne peut pas être inférieur à 1. De ce fait, $\mathcal{A} \rightsquigarrow^!_E \mathcal{A}'$ est convergente. □

Remarque. Les conditions associées au nouveau type d'équations peuvent être vues comme un paramètre contrôlant la finesse de l'approximation. En effet, sur notre exemple, l'approximation sera plus fine si $E = \{x = x + 1 \Leftarrow x > 4\}$, car les états équivalents seront fusionnés plus tard dans la complétion.

Étape de *widening*. Comme expliqué dans la sous-section précédente, chaque ensemble d'états contenant le terme $1 + 2$ devrait également contenir le terme 3, c'est pourquoi on effectue une étape d'évaluation de l'automate. Cependant, de la même manière qu'une étape de complétion rajoutant de nouvelles transitions à évaluer, cette propriété peut également être invalidée par la fusion d'états produite par l'abstraction équationnelle. En effet, cette fusion d'états va modifier certaines transitions du LTA. Nous devons donc effectuer une étape d'évaluation après la fusion par équation.

EXEMPLE *5.25*
Après la fusion de q_4 et q_6, on obtient :

$$\begin{aligned} Merge(\Delta_2, q_4, q_6) = eval(\Delta_1) \cup \{ \quad & q_3 + q_{[1,1]} \to q_4, f(q_4) \to q_5, \\ & cons(q_3, q_5) \to q'_2, q'_2 \to q_2, \\ & [2,3] \to q_4, q_4 + q_{[1,1]} \to q_4, \\ & f(q_4) \to q_7, cons(q_4, q_7) \to q'_5, \\ & q'_5 \to q_5, [3,4] \to q_4 \ \}. \end{aligned}$$

La transition $q_4 + q_{[1,1]} \to q_4$ doit être évaluée. La première itération de cette évaluation va évaluer le terme $[3,4] + [1,1]$, ce qui va ajouter la transition $[4,5] \to q_4$. Étant donné qu'un nouveau terme est reconnu par l'état q_4, la deuxième itération va évaluer le terme $[4,5]+[1,1]$ reconnu par la transition $q_4+q_{[1,1]} \to q_4$. Comme un nouvel élément du treillis sera toujours ajouté au langage de l'état q_4, ce schéma sera répété indéfiniment durant les étapes d'évaluations suivantes, et l'évaluation de l'automate va donc diverger. ◀

Comme la fonction $eval$ est définie comme un point-fixe de $propag$, son exécution ne peut pas converger sans l'application d'un opérateur de *widening* $\nabla_\Lambda : \Lambda \times \Lambda \mapsto \Lambda$. Cette méthode est classiquement utilisée dans le domaine de l'interprétation abstraite ([Cousot et Cousot, 1977]), afin de pouvoir calculer des sur-approximations de point-fixes (voir section 2.6.3). Cet opérateur de *widening* peut être paramétré afin de contrôler la finesse de l'abstraction, en choisissant par exemple le nombre d'étapes d'itérations de $propag$ à partir duquel il est appliqué, ou encore à partir d'un certain nombre de transitions $\lambda_i \to q$ s'ajoutant sur le même état q (voir figure 5.8). Ce paramétrage est important car selon la finesse de l'abstraction, certaines propriétés pourront être moins facilement vérifiées. Par exemple, imaginons que l'on veuille prouver que langage de l'état q_1 de la figure 5.8 ne contient pas les termes $[2,2]$ et $[4,4]$, on constate que l'opérateur de *widening* fusionnant dès la première étape ($\nabla_\Lambda(1)$) ne permet pas de vérifier cette propriété, alors que l'abstraction plus fine $\nabla_\Lambda(3)$ permet de la vérifier).

EXEMPLE *5.26*
Il est nécessaire d'appliquer un opérateur de widening ∇_Λ sur notre exemple fil rouge afin que l'évaluation converge en choisissant par exemple qu'il s'applique après 3 étapes d'itérations de la fonction $propag$. On a, au bout de trois étapes, les λ-transitions suivantes qui sont ajoutées sur q_4 : $[2,3] \to q_4$, $[3,4] \to q_4$, $[4,5] \to q_4$. Après application de ∇_Λ, la transition $[4,5] \to q_4$ est remplacée par $[4,+\infty[\to q_4$. Le résultat de la prochaine évaluation de la transition $q_4 + q_{[1,1]} \to q_4$ sera donc reconnu par la transition $[4,+\infty[\to q_4$ et l'évaluation va terminer. ◀

5.4.4 Algorithme complet de complétion pour LTA et preuve de correction

Nous pouvons désormais lister, tout d'abord de manière informelle, les différentes étapes de l'algorithme de complétion pour les LTA, dont chaque étape se déroule comme suit pour un automate $\mathcal{A}^i_{\mathcal{R},E}$ courant, un *CTRS* $\mathcal{R}$ et un ensemble E d'équations :

---

1. Étape d'évaluation de $\mathcal{A}^i_{\mathcal{R},E}$, et son opération de $widening$ si nécessaire $\Rightarrow$ calcul de $eval(\mathcal{A}^i_{\mathcal{R},E})$.

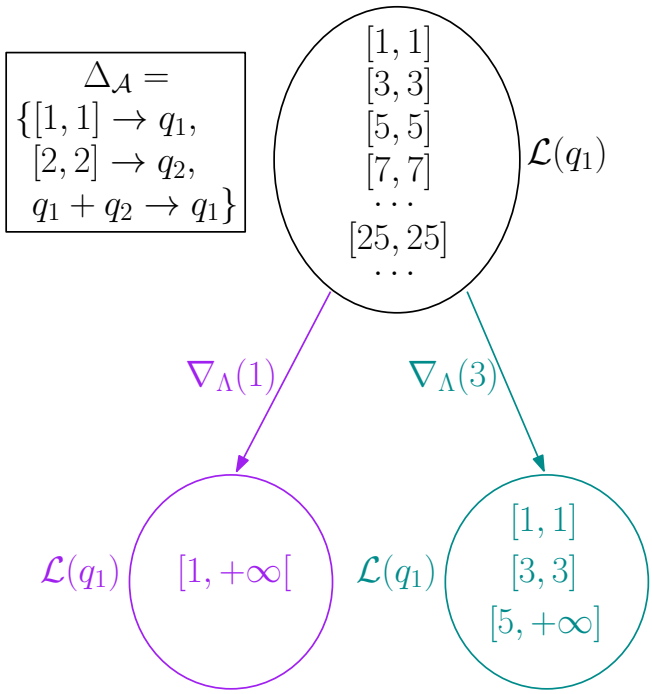


FIGURE 5.8 — Opérateur de *widening* pour faire converger l'évaluation. On note $\nabla_\Lambda(x)$ lorsque l'opérateur ∇_Λ est appliqué à la xme étape d'évaluation.

2. Calcul de l'automate successeur. On obtient alors $\mathcal{C}_{\mathcal{R}}(eval(\mathcal{A}^i_{\mathcal{R},E}))$.
   Soit, pour chaque règle $l \to r \Leftarrow c_1 \wedge \ldots \wedge c_n \in \mathcal{R}$:
   a) Calcul de toutes les substitutions possibles σ telles que $l\sigma \to^*_\Delta q$,
   b) Restriction des substitutions par le solveur. On obtient alors un nouvel ensemble de substitutions σ',
   c) Ajout des nouvelles transitions normalisées, soit $Norm(r\sigma' \to q)$.
3. Autre étape d'évaluation (avec *widening* si nécessaire) :
   calcul de $eval(\mathcal{C}_{\mathcal{R}}(eval(\mathcal{A}^i_{\mathcal{R},E})))$.
4. Étape d'abstraction par équations $\Rightarrow$ calcul de $\mathcal{A}^{i+1}_{\mathcal{R},E}$:
   – Équations classiques sur $\mathcal{T}(\mathcal{F}_\circ)$,
   – Équations conditionnelles sur $\mathcal{T}(\mathcal{F}_\bullet)$.

Donnons maintenant la définition formelle de cet algorithme.

DÉFINITION 5.4.12 (Algorithme de complétion pour LTA)
Soient $\mathcal{A}$ un LTA, $\mathcal{R}$ un CTRS et E un ensemble d'équations.
– $\mathcal{A}^0_{\mathcal{R},E} = \mathcal{A}$,
– *Répéter* $\mathcal{A}^{n+1}_{\mathcal{R},E} = \mathcal{A}'$ *avec* $eval(\mathcal{C}_{\mathcal{R}}(\mathcal{A}^n_{\mathcal{R},E})) \rightsquigarrow^!_E \mathcal{A}''$ *et* $eval(\mathcal{A}'') = \mathcal{A}'$,
– *Jusqu'à ce qu'un point-fixe* $\mathcal{A}^*_{\mathcal{R},E} = \mathcal{A}^k_{\mathcal{R},E} = \mathcal{A}^{k+1}_{\mathcal{R},E}$ *(avec* $k \in \mathbb{N}$*) soit atteint.*

Remarque. On peut distinguer dans cette complétion deux dimensions infinies. La première est due à l'espace d'états infini, et la deuxième est due au domaine concret ou

abstrait infini (ex : $\mathbb{Z}$). Le comportement infini du système (l'espace d'états infini) est capturé grâce aux équations, et le comportement infini dérivé de l'application des opérations du treillis (comme par exemple $x \to x+1$) est capturé par l'opération de *widening* inclue dans l'étape d'évaluation. En effet, ce comportement infini est engendré par des λ-transitions ajoutées au fur et à mesure, dont la valeur de leurs éléments évolue de façon croissante (ou décroissante), comme par exemple, la séquence croissante $[0,2] \to q$, $[2,4] \to q$, $[4,6] \to q$, ... Dans ce cas précis nous devons appliquer un opérateur de *widening* (ici $[0,+\infty[$) pour garantir la terminaison.

Mais une séquence infinie croissante (ou décroissante) de λ-transitions est nécessairement générée par une opération prédéfinie du treillis utilisée dans les règles de réécriture. Par exemple, la séquence croissante décrite ci-dessus est nécessairement due à une règle de réécriture de la forme $u(\ldots,\mathbf{x},\ldots) \to v(\ldots,\mathbf{x+2},\ldots)$. Si l'on a la substitution $x \mapsto q_1$, et la transition $[2,2] \to q_2$, alors la transition $q_1 + q_2 \to q_3$ sera ajoutée, et tant que cette règle de réécriture entraine un comportement infini (ajouter 2 indéfiniment), il y aura la séquence infinie $q_3 + q_2 \to q_4$, $q_4 + q_2 \to q_5$, *etc.* Pour résoudre ce problème, il est nécessaire d'utiliser une équation de la forme $x = x + 2$. Alors, q_1 et q_3 sont fusionnés et on a une transition $q_1 + q_2 \to q_1$ entrainant une évaluation infinie, abstraite grâce à l'opération de *widening* de l'étape d'évaluation. Pour résumer, une séquence infinie de λ-transitions est forcément générée par une opération interprétable, et étant donné que les transitions d'un LTA contenant des opérations doivent être évaluées, ce comportement infini est toujours résolu *durant l'étape d'évaluation*, comme nous allons le voir dans le théorème suivant.

THÉORÈME 5.4.13
Soient $\mathcal{R}$ un système de réécriture conditionnel et $\mathcal{A}^0_{\mathcal{R},E}$ un LTA initial. Durant l'algorithme de complétion visant à calculer $\mathcal{A}^*_{\mathcal{R},E}$, à chaque étape de calcul de l'automate successeur $\mathcal{A}^{i+1}_{\mathcal{R},E}$, l'opérateur de widening est appliqué uniquement durant l'étape d'évaluation, i.e. durant le calcul de $eval(\mathcal{A}^i_{\mathcal{R},E})$.

Démonstration. Selon sa définition (dans le paragraphe "Étape de *widening*"), l'opérateur de *widening* est appliqué seulement sur les lambdas-transitions, afin d'abstraire et capturer des lambdas-transitions s'ajoutant à l'infini. Durant la complétion, des transitions sont ajoutées à seulement 2 étapes :

1. le calcul de $\mathcal{C}_{\mathcal{R}}(\mathcal{A}^i_{\mathcal{R},E})$, et
2. l'étape d'évaluation.

1. Le calcul de $\mathcal{C}_{\mathcal{R}}(\mathcal{A}^i_{\mathcal{R},E})$ est effectué en appliquant le système de réécriture $\mathcal{R}$ sur l'automate $\mathcal{A}^i_{\mathcal{R},E}$. Selon la définition d'une étape de complétion (définition 5.4.7), pour toute règle de réécriture $l \to r \Leftarrow Cond \in \mathcal{R}$, si $l\sigma \to^*_{\mathcal{A}^i_{\mathcal{R},E}} q$, alors on ajoute $r\sigma' \to^* q$ dans $\mathcal{A}^i_{\mathcal{R},E}$, tel que σ' soit la restriction de σ respectant les conditions $Cond$. Cette partie droite ($r\sigma' \to^* q$) peut alors rajouter des éléments du domaine abstrait pouvant être soit un élément du treillis, soit un opérateur. Seul l'ajout d'un élément du treillis donne lieu à un ajout d'une λ-transition.

Or, si la partie droite de la règle permet d'ajouter une λ-transition, c'est qu'il y a un élément du domaine concret dans cette partie droite, mais il est de ce fait constant (*i.e.* la règle ajoute toujours le même élément du domaine concret comme dans la règle $x \to x+1$). Il y aura donc toujours la même transition à ajouter et donc pas de comportement infini à capturer par un *widening*.

En effet, soient c_i des éléments du domaine concret et (op_j) des opérateurs, et soit la partie droite de la règle r contenant des opérateurs et des éléments du domaine concret, alors les transitions ajoutées seront de la forme $c_i \to q_i$ et $op_j(q_1, ..., q_n) \to q_j$. Les seules lambdas-transitions à ajouter sont les transitions $c_i \to q_i$, et étant donné que la règle de réécriture ne change pas, il suffit d'ajouter une seule fois ces transitions dans l'automate. Comme ces règles seront déjà présentes dans l'automate, elles ne s'ajouteront pas à la prochaine application de la règle. Par exemple, pour la règle $x \to x + 1$, la λ-transition $[1, 1] \to q$ est ajoutée dans l'automate et n'a besoin d'être ajoutée qu'une seule fois (car déjà présente dans l'automate lors des futures applications de la règle).

2. Il n'y a donc pas de comportement infini à capturer au niveau des λ-transitions durant le calcul de $\mathcal{C}_{\mathcal{R}}(\mathcal{A}^i_{\mathcal{R},E})$. Ce comportement infini intervient donc durant l'étape d'évaluation. En effet, les opérations ajoutées durant le calcul de $\mathcal{C}_{\mathcal{R}}(\mathcal{A}^i_{\mathcal{R},E})$ auront besoin d'être évaluées, et leur évaluation peut donner lieu à un ajout infini de λ-transitions, comme expliqué dans le paragraphe "Étape de *widening*", notamment dans l'exemple 5.25. □

EXEMPLE *5.27*
Dans notre exemple fil rouge, grâce à l'opération de *widening* effectuée à l'étape d'évaluation précédente, il n'y a plus de transitions à ajouter à l'automate et la complétion termine. Nous avons alors un LTA point-fixe représentant une sur-approximation de l'ensemble des états accessibles :

$$\begin{aligned} eval(\Delta_2) = \{ \quad & [0,2] \to q_1, f(q_1) \to q_2, \\ & [1,2] \to q_3, [1,1] \to q_{[1,1]}, \\ & q_3 + q_{[1,1]} \to q_4, f(q_4) \to q_5, \\ & cons(q_3, q_5) \to q'_2, q'_2 \to q_2, [2,3] \to q_4, \\ & q_3 + q_{[1,1]} \to q_4, f(q_4) \to q_5, \\ & cons(q_3, q_5) \to q'_2, q'_2 \to q_2, \\ & [2,3] \to q_4, q_4 + q_{[1,1]} \to q_4, \\ & f(q_4) \to q_7, cons(q_4, q_7) \to q'_5, \\ & q'_5 \to q_5, [3,4] \to q_4, [4, +\infty[\to q_4 \ \}. \end{aligned}$$

◀

THÉORÈME 5.4.14 (Correction)
Soit $\mathcal{R}$ un CTRS linéaire gauche, $\mathcal{A}$ le LTA initial et E un ensemble d'équations (conditionnelles) sur $\mathcal{T}(\mathcal{F})$. Soit $\mathcal{A}^*_{\mathcal{R},E}$ le LTA point-fixe calculé par complétion s'il existe, alors on a $\mathcal{L}(\mathcal{A}^*_{\mathcal{R},E}) \supseteq \mathcal{R}^*(\mathcal{L}(\mathcal{A}))$.

Démonstration. Démontrons tout d'abord que $\mathcal{L}(\mathcal{A}^*_{\mathcal{R},E}) \supseteq \mathcal{L}(\mathcal{A})$. Par définition, la complétion n'ajoute que des transitions à $\mathcal{A}$. Ainsi, on a trivialement $\mathcal{L}(\mathcal{A}^1_{\mathcal{R}}) \supseteq \mathcal{L}(\mathcal{A})$. Grâce au théorème 5.4.11, on sait également que $\mathcal{A}^1_{\mathcal{R},E}$, la transformation de $\mathcal{A}^1_{\mathcal{R}}$ par fusion d'états équivalents selon l'ensemble d'équations E, est telle que $\mathcal{L}(\mathcal{A}^1_{\mathcal{R},E}) \supseteq \mathcal{L}(\mathcal{A}^1_{\mathcal{R}})$. De ce fait, par transitivité de $\supseteq$, on a $\mathcal{L}(\mathcal{A}^1_{\mathcal{R},E}) \supseteq \mathcal{L}(\mathcal{A})$. En suivant ce même raisonnement de manière successive pour $\mathcal{A}^2_{\mathcal{R},E}, \mathcal{A}^3_{\mathcal{R},E}, \mathcal{A}^4_{\mathcal{R},E}, \ldots$, on obtient alors $\mathcal{L}(\mathcal{A}^*_{\mathcal{R},E}) \supseteq \mathcal{L}(\mathcal{A})$.

La prochaine étape de la preuve consiste à démontrer que pour tout terme $s \in \mathcal{L}(\mathcal{A})$, si $s \to^*_{\mathcal{R}} t$ alors $t \in \mathcal{L}(\mathcal{A}^*_{\mathcal{R},E})$. Tout d'abord, on peut noter que les états finaux sont préservés, par définition de l'application de E, *i.e.* si q est un état final de $\mathcal{A}$ alors si $\mathcal{A}'$ est le LTA pour lequel E est appliqué sur $\mathcal{A}$ et pour lequel q a été renommé en q', alors q' est un état final de $\mathcal{A}'$. Ainsi il reste seulement à prouver que pour tout terme $s \in \mathcal{L}(\mathcal{A}, q)$, si $s \to^*_{\mathcal{R}} t$ alors $\exists q' : t \in \mathcal{L}(\mathcal{A}^*_{\mathcal{R},E}, q')$. Nous procédons par récurrence sur la

longueur de $\rightarrow^*_{\mathcal{R}}$:

- Si cette longueur est nulle, alors $s \rightarrow^*_{\mathcal{R}} s$ et on peut trivialement supposer que $s \in \mathcal{L}(\mathcal{A}^*_{\mathcal{R},E}, q')$.
- Supposons désormais que cette propriété est vraie pour toute chaine de réécriture de longueur inférieure ou égale à n. Nous devons alors prouver que la propriété reste vraie pour une chaine de réécriture de longueur inférieure ou égale à $n+1$. Supposons que $s \rightarrow^n_{\mathcal{R}} s' \rightarrow_{\mathcal{R}} t$. En utilisant l'hypothèse de récurrence, on obtient alors $s' \in \mathcal{L}(\mathcal{A}^*_{\mathcal{R},E}, q')$. Il reste alors à prouver que $t \in \mathcal{L}(\mathcal{A}^*_{\mathcal{R},E}, q')$ peut être déduit de $s' \rightarrow_{\mathcal{R}} t$. Comme on a $s' \rightarrow_{\mathcal{R}} t$, on sait qu'il existe une règle de réécriture $l \rightarrow r \Leftarrow c_1 \wedge \ldots \wedge c_m$, une position p et une substitution $\mu : \mathcal{X} \mapsto \mathcal{T}(\mathcal{F})$ telles que $s' = s'[l\mu]_p \rightarrow_{\mathcal{R}} eval(s'[r\mu]_p) = t$ et pour tout $j \in [1,m]$, $c_j\mu = true$. Comme $s' \in \mathcal{L}(\mathcal{A}^*_{\mathcal{R},E}, q')$, $s'[l\mu]_p \rightarrow^*_{\mathcal{A}^*_{\mathcal{R},E}} q'$ et par définition du langage d'un LTA, on peut alors déduire qu'il existe un s'' tel que $s' \sqsubseteq s''$ et $s'' \rightarrow^*_{\mathcal{A}^*_{\mathcal{R},E}} q'$. Par conséquent, on obtient $s''[l\mu]_p \rightarrow^*_{\mathcal{A}^*_{\mathcal{R},E}} q'$ et, par définition de la relation de transition induite par les LTA, qu'il existe un état q'' tel que $l\mu \rightarrow^*_{\mathcal{A}^*_{\mathcal{R},E}} q''$ et $s''[q'']_p \rightarrow^*_{\mathcal{A}^*_{\mathcal{R},E}} q'$. Soient $Var(l) = \{x_1, \ldots, x_k\}$, $l = l[x_1, \ldots, x_k]$ et $t_1, \ldots, t_k \in \mathcal{T}(\mathcal{F})$ tels que $\mu = \{x_1 \mapsto t_1, \ldots, x_k \mapsto t_k\}$. Comme $l\mu = l[t_1, \ldots, t_k] \rightarrow^*_{\mathcal{A}^*_{\mathcal{R},E}} q''$, on sait qu'il existe des états $q_1, \ldots, q_k$ tels que $\forall i \in [1,k]$, $t_i \rightarrow^*_{\mathcal{A}^*_{\mathcal{R},E}} q_i$ et $l[q_1, \ldots, q_k] \rightarrow^*_{\mathcal{A}^*_{\mathcal{R},E}} q''$. Soit $\sigma = \{x_1 \mapsto q_1, \ldots, x_k \mapsto q_k\}$, on a donc $l\sigma \rightarrow^*_{\mathcal{A}^*_{\mathcal{R},E}} q''$ grâce à la propriété de linéarité gauche.
  On a alors $\mu = \{x_1 \mapsto t_1, \ldots, x_k \mapsto t_k\}$ et $\forall i \in [1,k]$, $t_i \rightarrow^*_{\mathcal{A}^*_{\mathcal{R},E}} q_i$. Comme pour tout $j \in [1,m]$, on a $c_j\mu = true$, par définition des prédicats, tous les t_i sont des termes interprétables (ou bien $c_j\mu$ aurait été égal à faux). Donc pour tout $i \in [1,k]$ il existe $\lambda_i \in \Lambda$ tel que $eval(t_i) = \lambda_i$. Soit μ' une substitution $\{x_1 \mapsto \lambda_1, \ldots, x_k \mapsto \lambda_k\}$, alors on peut déduire que pour tout $j \in [1,m]$, $c_j\mu' = true$. Grâce à l'étape d'évaluation, on déduit alors que pour tout $i \in [1,k]$, si $t_i \rightarrow^*_{\mathcal{A}^*_{\mathcal{R},E}} q_i$, alors $eval(t_i) = \lambda_i \rightarrow^*_{\mathcal{A}^*_{\mathcal{R},E}} q_i$. La propriété du solveur permet d'établir que $Solve(\sigma, \mathcal{A}, c_1 \wedge \cdots \wedge c_m) = \bigcup_{\lambda_1 \rightarrow^*_{\mathcal{A}} q_1, \ldots, \lambda_k \rightarrow^*_{\mathcal{A}} q_k} Solve_\Lambda(\sigma/c, \langle \lambda_i | i \in S_c \rangle, c)$. On peut alors déduire que pour tout $j \in [1,m]$ $c_i\sigma = true$ car $c_i\mu' = true$. Comme $\mathcal{A}^*_{\mathcal{R},E}$ est un point-fixe obtenu par complétion, on peut déduire grâce à $l\sigma \rightarrow^*_{\mathcal{A}^*_{\mathcal{R},E}} q''$ et le fait que pour tout $j \in [1,m]$, $c_j\sigma = true$, que $r\sigma \rightarrow^*_{\mathcal{A}^*_{\mathcal{R},E}} q''$. De plus, comme $\forall i \in [1,k]$, $t_i \rightarrow^*_{\mathcal{A}^*_{\mathcal{R},E}} q_i$, alors $r\mu \rightarrow^*_{\mathcal{A}^*_{\mathcal{R},E}} q''$. Comme par ailleurs on a $s''[q'']_p \rightarrow^*_{\mathcal{A}^*_{\mathcal{R},E}} q'$, alors on peut déduire que $s''[r\mu]_p \rightarrow^*_{\mathcal{A}^*_{\mathcal{R},E}} q'$. Comme $s' \sqsubseteq s''$, cela signifie par définition que $eval(s') \sqsubseteq eval(s'')$. Pour conclure, comme $s''[r\mu]_p \rightarrow^*_{\mathcal{A}^*_{\mathcal{R},E}} q'$ et $eval(s') \sqsubseteq eval(s'')$, on peut déduire que $t = eval(s'[r\mu]_p) \rightarrow^*_{\mathcal{A}^*_{\mathcal{R},E}} q'$, ainsi $t \in \mathcal{L}(\mathcal{A}^*_{\mathcal{R},E}, q')$.

□

5.5 Complétion pour LTA déroulée sur un exemple

L'exemple fil rouge des sections précédentes était situé dans le domaine des entiers relatifs, abstrait par l'ensemble des intervalles. Nous pouvons ainsi par exemple simplifier le traitement et la modélisation des entiers dans la vérification de programmes Java. Nous voulons ici montrer que les LTA permettent de simplifier la modélisation d'autres types de données présentes habituellement dans les programmes, comme les chaines de caractères. Nous allons ici dérouler un exemple dont le domaine concret infini $\mathcal{C}$

sera l'ensemble de toutes les chaines de caractères possibles, de taille potentiellement infinie, construites à partir d'un alphabet α.

EXEMPLE *5.28*
Si $\alpha = \{a, b\}$, alors $\mathcal{C} = \{aa, ab, bb, aba, \ldots, aaaabbbababbabba, \ldots\}$. ◀

Dans le cas de la vérification d'un programme Java, α peut alors être l'ensemble des caractères ASCII. Le domaine abstrait choisi correspondant est celui de l'ensemble des expressions régulières, soit l'ensemble des expressions $\langle E \rangle$ avec $E = EE \mid (E)^* \mid (E|E) \mid a \in \alpha$, où $(E)^*$ signifie une ou plusieurs successions de l'expression E, et où $(E|E)$ signifie le choix entre deux expressions (*ou* exclusif).

EXEMPLE *5.29*
L'expression $\langle a^*(b|c) \rangle$ permet de représenter l'ensemble infini de mots qui commencent par zéro ou plusieurs *'a'* suivis par un *'b'* ou un *'c'*. ◀

La seule opération –interprétable grâce à la fonction $eval$– que nous allons utiliser ici est la concaténation de deux mots, représentée par le symbole "^". Soit α un alphabet quelconque, on a donc $\mathcal{F}^{\#}_{\bullet} = \{\langle E \rangle \mid E = EE \mid (E)^* \mid (E|E) \mid a \in \alpha\} \cup \{\hat{}\}$. Nous allons également utiliser des opérations définies pour l'ensemble de contraintes $\mathcal{P}$: $n(a, x)$ qui va renvoyer le nombre de caractères *'a'* présents dans la chaine de caractère x, et les opérateurs classiques de comparaisons tels que $<$ ou $>$.

EXEMPLE *5.30*
Par exemple, $n(r, arbre) > 1)$ appartient à $\mathcal{P}$ et est évalué à $vrai$. Dans le cas des expressions régulières, $eval^{\#}(n(r, \langle (arb)^* re \rangle) < 3)) = faux$.

◀

Dans le cas de la complétion classique, un mot pourrait être représenté par une liste de lettres. Par exemple, le mot "*arbre*" serait représenté par le terme $cons(a, cons(r, cons(b, cons(r, cons(e, nil)))))$. Dans ce cas, la concaténation de deux mots correspond à la concaténation de deux listes, qui est modélisée par l'ensemble de règles de réécriture suivant :

$$\mathcal{R} = \{append(cons(x, y), z) \rightarrow cons(x, append(y, z)), append(nil, x) \rightarrow x\}.$$

Pour la concaténation de deux chaines de caractères, soit $append(m_1, m_2)$, on va appliquer les règles de réécritures ci-dessus $(taille(m_1) - 1) \times 2 + 1$ fois. Donc pour la concaténation de deux chaines de caractères dont la première est de longueur 15, on va alors appliquer 29 fois les règles de réécriture.

Mais dans le cas des LTA, la fonction $eval$ va pouvoir interpréter en une seule étape la concaténation de deux mots, au lieu d'appliquer un grand nombre de fois les règles de réécriture énoncées ci-dessus.

EXEMPLE *5.31*
On a alors $eval(comple\hat{}tion) = completion$. Dans le cas du domaine abstrait, on a par exemple $eval^{\#}(\langle (arb)^* \rangle \hat{} \langle (re|uste) \rangle) = \langle (arb)^*(re|uste) \rangle$. ◀

Soit $\mathcal{A}_0 = \langle \mathcal{F} = \mathcal{F}_\circ \cup \mathcal{F}^{\#}_{\bullet}, \mathcal{Q}_0, \mathcal{Q}_{F_0}, \Delta_0 \rangle$ un LTA tel que $\mathcal{F}_\circ = \{f_2\}$ et $\mathcal{F}^{\#}_{\bullet} = \Lambda \cup \{\hat{}\}$, $\mathcal{Q}_0 = \{q_1, q_2, q_f\}$, $\mathcal{Q}_{F_0} = \{q_f\}$, et

$\Delta_0 = \{ \quad \langle a^* \rangle \to q_1, \langle b^* \rangle \to q_2,$
$\quad f(q_1, q_2) \to q_f \ \}$

Soit $\mathcal{R} = \{f(x, y) \to f(x^\wedge c, y^\wedge d\}$ un système de réécriture et $E = \{x = x^\wedge c \Leftarrow n(c, x) > 0, x = x^\wedge d \Leftarrow n(d, x) > 0\}$ un ensemble d'équations définies pour les LTA. Nous allons alors appliquer l'algorithme de complétion pour les LTA afin de calculer une surapproximation de l'ensemble des accessibles.

Calculons le LTA $\mathcal{A}_1$ obtenu après une étape de complétion. Ici nous remarquons que nous devons ajouter $Norm(f(q_1{}^\wedge c, q_2{}^\wedge d)$ à l'automate. On a donc :

$$\begin{aligned}\Delta_1 = \{ \quad & \langle a^* \rangle \to q_1, \langle b^* \rangle \to q_2,\\ & f(q_1, q_2) \to q_f,\\ & \langle c \rangle \to q_c, \langle d \rangle \to q_d,\\ & q_1{}^\wedge q_c \to q_3, q_2{}^\wedge q_d \to q_4,\\ & f(q_3, q_4) \to q'_f, q'_f \to q_f \ \}\end{aligned}$$

Après l'étape d'évaluation, nous allons donc obtenir, en évaluant les deux transitions $q_1{}^\wedge q_c \to q_3$ et $q_2{}^\wedge q_d \to q_4$, les deux transitions supplémentaires suivantes :
$eval(\Delta_1) = \Delta_1 \cup \{\langle (a)^* c \rangle \to q_3, \langle (b)^* d \rangle \to q_4\}$.

Nous devons maintenant appliquer l'abstraction équationnelle, soit les deux équations $x = x^\wedge c \Leftarrow n(c, x) > 0$ et $x = x^\wedge d \Leftarrow n(d, x) > 0$. Nous pouvons ensuite remarquer que ces équations peuvent s'appliquer sur les deux transitions $q_1{}^\wedge q_c \to q_3$ et $q_2{}^\wedge q_d \to q_4$. Mais ces deux équations possèdent des conditions : $n(c, x) > 0$ et $n(d, x) > 0$. Ici, x correspond respectivement aux états q_1 et q_2, donc aux expressions régulières $\langle a^* \rangle$ et $\langle b^* \rangle$. Or $n(c, \langle a^* \rangle) = 0$ et $n(d, \langle b^* \rangle) = 0$, donc les conditions ne sont pas respectées. On ne peut donc pas encore appliquer les équations.

Une autre étape de complétion est donc effectuée. L'ensemble de transitions découlant de $Norm(f(q_3{}^\wedge c, q_4{}^\wedge d)$ est alors ajouté. On a alors :

$$\begin{aligned}\Delta_2 = eval(\Delta_1) \cup \{ \quad & q_3{}^\wedge q_c \to q_5, q_4{}^\wedge q_d \to q_6\\ & f(q_5, q_6) \to q''_f, q''_f \to q'_f \ \}\end{aligned}$$

Après évaluation des transitions $q_3{}^\wedge q_c \to q_5$ et $q_4{}^\wedge q_d \to q_6$, on obtient :
$eval(\Delta_2) = \Delta_2 \cup \{\langle (a)^* cc \rangle \to q_5, \langle (b)^* dd \rangle \to q_6\}$.

Nous constatons ensuite que les deux transitions $q_3{}^\wedge q_c \to q_5$ et $q_4{}^\wedge q_d \to q_6$ correspondent respectivement aux équations $x = x^\wedge c \Leftarrow n(c, x) > 0$ et $x = x^\wedge d \Leftarrow n(d, x) > 0$. De plus, x correspondant respectivement aux états q_3 et q_4, on a $n(c, \langle a^* c \rangle) = 1$ et $n(d, \langle b^* d \rangle) = 1$ (d'après les deux transitions $\langle (a)^* c \rangle \to q_3$ et $\langle (b)^* d \rangle \to q_4$ calculées après évaluation de Δ_1), donc les conditions sont respectées. Selon les équations, les états q_3 et q_5 doivent être fusionnés, ainsi que les états q_4 et q_6.

On a alors, après application de l'abstraction, l'ensemble de transitions suivant :

$$\begin{aligned}\rightsquigarrow^!_E (eval(\Delta_2)) = eval(\Delta_1) \cup \{ \quad & q_3{}^\wedge q_c \to q_3, q_4{}^\wedge q_d \to q_4\\ & \langle (a)^* cc \rangle \to q_3, \langle (b)^* dd \rangle \to q_4,\\ & f(q_3, q_4) \to q''_f, q''_f \to q'_f \ \}\end{aligned}$$

Nous devons ensuite à nouveau procéder à une étape d'évaluation de l'automate. Ici, les transitions à évaluer sont les transitions $q_3{}^\wedge q_c \to q_3$ et $q_4{}^\wedge q_d \to q_4$. On a $eval(q_3{}^\wedge q_c \to q_3) = \{\langle (a)^* ccc \rangle \to q_3\}$. Or, comme il y a un nouvel élément dans l'état q_3, l'évaluation se poursuit et $eval(q_3{}^\wedge q_c \to q_3) = \{\langle (a)^* ccc \rangle \to q_3, \langle (a)^* cccc \rangle \to q_3\}$. Comme un nouvel élément vient de s'ajouter dans q_3, on a donc $eval(q_3{}^\wedge q_c \to q_3) = \{\langle (a)^* ccc \rangle \to q_3, \langle (a)^* cccc \rangle \to q_3, \langle (a)^* ccccc \rangle \to q_3\}$ et ainsi de suite. L'évaluation ne va donc pas ter-

miner et ajouter indéfiniment un *'c'* à la chaine de caractère reconnue par q_3. Il en va de même pour l'évaluation de $q_4{}^\wedge q_d \to q_4$ qui va ajouter indéfiniment la lettre *'d'*.

Rappelons le principe du *widening* : à partir d'un certain nombre d'ajouts de λ-transitions sur un même état lors de l'étape d'évaluation, ces transitions sont fusionnées selon le widening fourni avec le treillis. Dans le cas des intervalles, on fusionne alors une séquence croissante d'intervalles comme $[1,2],[3,4],[5,6],\ldots$, en la remplaçant par l'intervalle $[1,+\infty[$. Dans le cas des chaines de caractères, la succession d'une même séquence de lettres est fusionnée à partir d'une étape donnée.

EXEMPLE *5.32*
La séquence $\langle ee\rangle, \langle eee\rangle, \langle eeee\rangle, \ldots$ peut être fusionnée en $\langle e^*\rangle$. ◀

Ici, si nous décidons d'appliquer le widening au bout de deux étapes, la séquence $\langle (a)^*ccc\rangle, \langle (a)^*cccc\rangle, \langle (a)^*ccccc\rangle$ reconnue par q_3 va donc être fusionnée en $\langle (a)^*(c)^*\rangle$, et la séquence $\langle (b)^*ddd\rangle, \langle (b)^*dddd\rangle, \langle (a)^*ddddd\rangle$ reconnue par q_4 sera fusionnée en $\langle (b)^*(d)^*\rangle$.

On obtient alors l'automate suivant :

$$\begin{aligned} eval(\rightsquigarrow_E^! (eval(\Delta_2))) = eval(\Delta_1) \cup \{ \quad & q_3{}^\wedge q_c \to q_3, q_4{}^\wedge q_d \to q_4 \\ & \langle (a)^*cc\rangle \to q_3, \langle (b)^*dd\rangle \to q_4, \\ & \langle (a)^*(c)^*\rangle \to q_3, \langle (b)^*(d)^*\rangle \to q_4, \\ & f(q_3,q_4) \to q''_f, q''_f \to q'_f \ \} \end{aligned}$$

Ainsi, l'évaluation termine. En essayant de calculer l'automate successeur par complétion, nous constatons que $f(q_3{}^\wedge c, q_4{}^\wedge d)$ est déjà reconnu par cet automate. L'algorithme de complétion termine donc également, et nous avons calculé une sur-approximation de l'ensemble des états accessibles.

5.6 Conclusion

Dans ce chapitre, nous avons proposé les *automates d'arbres à treillis*, nommés LTA, un nouveau type d'automates d'arbres dont les configurations peuvent être représentées par des termes *interprétables*, *i.e.* contenant les éléments d'un treillis et ses opérations associées. L'intérêt de tels automates est de calculer le résultat d'une opération arithmétique en une seule étape d'*évaluation* plutôt que d'appliquer un nombre élevé de règles de réécritures.

Nous avons ensuite adapté les différentes opérations des automates d'arbres classiques aux LTA, notamment la déterminisation. Certains langages reconnaissables par un LTA ne peuvent être reconnus par un automate déterministe. L'opération de déterminisation nécessite le *partionnement* de l'automate donné en entrée selon une partition du treillis utilisé. L'algorithme proposé retourne donc une *approximation*, dont la précision dépend de la finesse de la partition utilisée.

Puis enfin, notre principale contribution est le développement d'un nouvel algorithme de complétion adapté aux LTA. Cet algorithme est défini pour des systèmes de réécriture conditionnels (*CTRS*) intégrant les éléments du domaine *concret*, afin de les rendre indépendant du treillis *abstrait* utilisé dans l'automate. Cet algorithme nécessite alors l'utilisation d'un solveur afin de garantir la satisfiabilité des contraintes du système de réécriture. Une autre spécificité de cet algorithme est une étape d'évaluation permettant d'avoir l'ensemble des termes reconnus par l'automate, équipée d'un opérateur de *widening* associé au treillis afin de garantir la terminaison de cette étape. Une

bonne propriété de ce nouvel algorithme est qu'il est indépendant du treillis utilisé : il suffit qu'il soit unidimensionnel, atomique et équipé d'un solveur pour les prédicats du *CTRS*. N'importe quel treillis remplissant ces conditions peut être branché de façon transparente à cet algorithme de complétion. En effet, cette technique est *paramétrable* : l'algorithme est paramétré tout d'abord par le domaine concret utilisé, puis par le domaine abstrait correspondant, et enfin par le paramètre du *widening*. La précision du calcul des accessibles peut ainsi être améliorée en ajustant ces différents paramètres.

Il restait ensuite à prouver expérimentalement que ce nouveau codage pouvait considérablement améliorer la vérification des systèmes de réécriture utilisant un domaine infini, ou encore de l'arithmétique. Ce que nous avons illustré par une implémentation décrite au chapitre 6, pour la vérification des programmes $Java$. Nous avons implémenté TimbukLTA une adaptation de Timbuk (voir section 2.5) utilisant la complétion pour LTA, pour les entiers relatifs et le treillis des intervalles.

Nous avons vu que le treillis devait être unidimensionnel. En effet, étant reconnu par un état de l'automate, et correspondant ainsi à une seule variable du système de réécriture, il ne permet d'abstraire qu'une seule valeur. Il est donc inutile qu'il soit relationnel. Prenons par exemple le système de réécriture suivant : $\{f(x) \to f(x+1) \Leftarrow x > 2\}$. Il est inutile d'utiliser le treillis des polyèdres et ainsi d'avoir par exemple l'ensemble de transitions suivant : $\Delta = \{(A + B < 4) \to q_1, f(q_0) \to q_1\}$.

Cependant, l'intégration de treillis relationnels est possible dans l'algorithme de complétion. Mais cela implique que chaque élément du treillis doit être défini de façon globale, *i.e.* pour l'ensemble de l'automate, et non pas sur un seul état. Les contraintes sont définies sur les états de l'automates qui deviennent alors des variables, comme c'est le cas dans le chapitre 4. Imaginons par exemple l'ensemble de transitions Δ_0 suivant, qui serait fourni avec le polyèdre P_0 suivant :

$$\Delta_0 = \{f(X_0, X_1) \to X_3\} \text{ et } P_0 = \{X_0 + X_1 < 4\}.$$

Mais cela signifie que l'algorithme de complétion doit être adapté à ces nouveaux types d'automates munis d'un treillis relationnel, ce que nous verrons dans les perspectives de cette thèse (chapitre **Conclusion et perspectives**).

Implémentation de TimbukLTA et CopsterLTA et expérimentations

6

Sommaire

6.1 Introduction **181**

6.2 Préliminaires : fonctionnement de Copster **182**

6.2.1 Formalisation de la sémantique $Java$ par des règles de réécriture 182

6.2.2 Implémentation et exemple 188

6.3 Implémentation de CopsterLTA et TimbukLTA **195**

6.3.1 CopsterLTA 195

6.3.2 TimbukLTA 198

6.4 Expérimentations **201**

6.4.1 Un exemple introductif 201

6.4.2 Un exemple sans arithmétique : `Threads` 203

6.4.3 Un exemple avec arithmétique et méthode récursive : `Euclide` 205

6.4.4 Un exemple avec arithmétique et allocation d'objet : `FactoList` 206

6.5 Conclusion et travaux futurs **208**

"Le seul pêché est de ne pas se risquer pour suivre son désir."

Françoise Dolto

Résumé

Au chapitre précédent, nous avons décrit un nouveau type d'automates d'arbres, permettant d'intégrer et interpréter les éléments et opérations d'un domaine infini tel que les entiers relatifs, ainsi que le nouvel algorithme de complétion qui lui est associé.

Dans ce chapitre, nous allons décrire l'implémentation de cette méthode dans un nouvel outil appelé TimbukLTA. Après adaptation de l'outil Copster [Barré et al., 2009] à cette technique, afin de générer automatiquement un fichier Timbuk de règles de réécriture depuis un programme $Java$, nous prouverons ensuite l'efficacité de cette méthode par rapport à la complétion dite classique sur une série d'exemples.

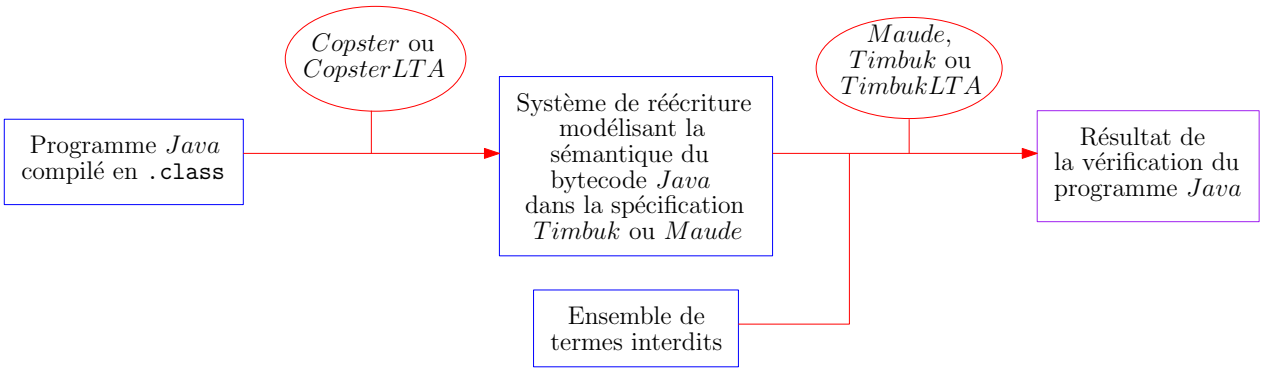


FIGURE 6.1 — Vérification d'un programme Java grâce à Copster et Timbuk.

6.1 Introduction

Nous avons vu lors du chapitre précédent l'intégration de treillis dans les automates d'arbres. En effet, les automates d'arbres à treillis (ou LTA), permettent de modéliser les éléments d'un domaine abstrait ainsi qu'interpréter directement les opérations sur ce domaine plutôt que de les implémenter par l'application successive de règles de réécriture. Nous avons alors définit un nouvel algorithme de complétion pour ces LTA. Dans ce chapitre, nous présentons l'implémentation de cette méthode, afin de montrer à quel point elle peut simplifier l'analyse des programmes *Java*.

Dans [Boichut et al., 2007], les auteurs ont développé une expression de la sémantique du bytecode *Java* sous forme de règles de réécriture, qui a ensuite été implémentée dans outil appelé Copster [Barré et al., 2009]. Cet outil permet de transformer un fichier *Java* `.class` en un système de réécriture. Le système de réécriture obtenu modélise exactement un sous-ensemble de la sémantique de la JVM (*Java* Virtual Machine : machine virtuelle *Java*) : essentiellement les types basiques, l'arithmétique, la création d'objets, la manipulation de champs, l'invocation des méthodes virtuelles, ainsi qu'un sous-ensemble de la librairie des *strings*. Cette modélisation consiste à réécrire, via le système de réécriture généré, un terme représentant l'état de la *JVM*. Le système de réécriture obtenu automatiquement grâce à Copster à partir du fichier *Java* `.class` est généré dans le format utilisé par Timbuk. Un fichier au format `Maude` peut également être généré. Le programme *Java* peut ainsi être vérifié dans Timbuk ou `Maude`, comme nous pouvons le voir sur la figure 6.1.

La principale difficulté de cette modélisation est de capturer et gérer les deux dimensions infinies qui peuvent apparaître dans un programme *Java*. En effet, le comportement infini de ces programmes peut être dû à un nombre non-borné d'appels de méthodes ou de création d'objets, ou simplement parce que le programme manipule des données dont le domaine n'est pas borné, comme par exemple les variables entières. Tandis que les multiples comportements infinis peuvent être sur-approchés grâce à la complétion (par exemple $a^n b^n$ peut être abstrait par $a^* b^*$), la manipulation des domaines infinis tels que les entiers nécessite des modélisations lourdes et de tailles importantes telles que les entiers de Peano, utilisés dans l'implémentation de Copster et Timbuk. Mais ce choix a un impact sur la taille des automates utilisés pour encoder les ensembles de configurations. Il a aussi un impact sur l'exécution de chaque opération arithmétique, qui peut alors exiger l'application de nombreuses règles de réécriture.

```
xsub(a,zero)            →  result(a)
xsub(zero,pred(a))      →  xsub(succ(zero),a)
xsub(zero,succ(a))      →  xsub(pred(zero),a)
xsub(pred(a),pred(b))   →  xsub(a,b)
xsub(pred(a),succ(b))   →  xsub(pred(pred(a)),b)
xsub(succ(a),pred(b))   →  xsub(succ(succ(a)),b)
xsub(succ(a),succ(b))   →  xsub(a,b)
```

$$\texttt{xsub(succ(succ(zero)), succ(zero))} \xrightarrow{1} \texttt{xsub(suc(zero),zero)} \xrightarrow{2} \texttt{result(succ(zero))}$$

FIGURE 6.2 — Règles de réécriture représentant la soustraction de deux entiers relatifs.

Nous l'avons constaté dans le cas de l'addition en figure 5.1 du chapitre précédent, et nous le voyons également en figure 6.2 pour les règles modélisant la soustraction.

L'utilisation des LTA peut considérablement simplifier ces opérations arithmétiques. En effet, dans nos travaux, les entiers relatifs et leurs opérations sont encodés directement dans l'alphabet de l'automate. Dans un tel contexte, la succession d'applications de règles de réécriture est remplacée par une étape d'évaluation.

Ainsi nous verrons en section 6.3.1 de quelle manière les règles de réécriture de la figure 6.2 peuvent être simplifiées par une seule règle, appliquée une seule fois. Dans le cas de la complétion classique, le terme représentant la soustraction $255-210$ nécessitait l'application 255 fois des règles de réécriture de la figure 6.2. Ici, avec la complétion LTA, l'étape d'évaluation de la complétion LTA va compiler le résultat de la soustraction et ajouter le terme résultat dans le langage de l'automate courant, sans réécrire aucun terme.

C'est pourquoi nous avons implémenté un outil appelé TimbukLTA adapté de Timbuk, afin de prouver l'efficacité la complétion LTA. Nous avons également modifié Copster afin qu'il s'adapte à la complétion LTA, afin d'orienter les expérimentations et comparaisons sur des cas concrets tels que les programmes $Java$.

Nous allons tout d'abord décrire le fonctionnement de l'outil Copster, puis nous détaillerons ensuite les modifications que nous lui avons apporté, ainsi que l'implémentation de TimbukLTA. Nous terminerons ensuite par des expérimentations et comparaisons.

6.2 Préliminaires : fonctionnement de Copster

6.2.1 Formalisation de la sémantique $Java$ par des règles de réécriture

Une description précise de la sémantique $Java$ transformée en règles de réécriture peut être trouvée dans [Boichut et al., 2007]. Copster permet de transformer un programme $Java$ en règles de réécritures modélisant une partie significative de la sémantique du bytecode $Java$: les piles, les *frames*, les objets, les références, les méthodes, les tas, les entiers, et également le *multithread*.

Formalisation des états et des *frames*.

L'état d'une JVM est un terme de la forme $IO(st, in, out)$ où st est l'état d'un programme, in un flux d'entrée et out un flux de sortie.
Un état d'un programme $Java$ est un terme de la forme $state(f, fs, h, k)$, avec :

1. f la frame d'exécution courante, qui permet de donner les informations sur la méthode courante exécutée,
2. fs la pile des frames appelées par le programme :
   quand une nouvelle méthode est appelée, la frame courante est stockée dans la pile des frames et une nouvelle frame courante est appelée,
3. h un tas, pour stocker des objets et des tableaux, *i.e.* toutes les informations qui ne sont pas locales à l'exécution d'une méthode,
4. k un tas statique, qui permet de stocker les valeurs des champs statiques, *i.e.* les valeurs qui sont partagées par tous les objets d'une même classe.

Une frame est un terme de la forme $frame(m, pc, s, l)$, avec :

1. m un couple (n, c) où n est le nom de la méthode et c sa classe (par exemple, `name(foo,A)` représente la méthode `foo` de la classe `A`),
2. pc le point de programme courant. Soit x le nombre de points du programme p, le point de programme courant est une constante (*i.e.* d'arité 0) de l'alphabet $\mathcal{F}_{pc}(p) = \{pp0, pp1, \ldots, ppx\}$.
3. s la pile des opérandes (par exemple $stack(a, stack(b, nil))$ si les deux valeurs a et b ont été mises en haut de la pile par un `push`), et
4. l le tableau des variables locales.

La référence d'un objet est encodée par le terme `loc(c,a)` où `c` est la classe de l'objet référencé, et `a` un entier permettant de donner la référence de l'objet. Par exemple, `loc(foo,succ(zero))` est une référence pointant sur l'objet localisé à l'index 1 du tas de la classe `foo`.

Formalisation des instructions.

Pour un point de programme pc donné, dans une méthode `m` donnée, Copster construit un terme $xframe$ très similaire au terme $frame$, mais dont l'instruction courante est indiquée explicitement, afin de compiler des étapes intermédiaires.

En effet, le rôle des règles $frame$ est simplement d'expliciter l'instruction bytecode de chaque point de programme. Elles *dépendent* donc du programme que l'on veut modéliser. À l'inverse, les règles $xframe$ que les $frame$ permettent d'appeler sont des règles génériques représentant la sémantique des instructions du bytecode $Java$, et sont donc *indépendantes* du programme à modéliser. Comme nous allons le voir dans la suite, les règles $xframe$ sont générées par Copster quelque soit le programme, alors que les règles $frame$ vont être construites selon les instructions que le programme va appeler.

EXEMPLE *6.1*
Considérons l'opération arithmétique "$300 - 400$". Dans Copster, comme décrit dans [Boichut et al., 2007], cette opération est représentée par le terme $xsub(succ^{300}(zero), succ^{400}(zero))$, qui se réduit grâce aux règles représentant la soustraction (décrites en

figure 6.2), et qui devront être appliquées 300 fois. Cela signifie qu'au point de programme $pc = pp1$ de la méthode $m = name(foo, A)$, s'il y a le bytecode sub alors la $frame$ bascule en $xframe$ pour pouvoir calculer la soustraction, *i.e.* appliquer la règle :

$$frame(name(foo, A), pp1, s, l) \rightarrow xframe(sub, name(foo, A), pp1, s, l).$$

Pour calculer le résultat de la soustraction des deux premiers éléments de la pile, nous devons appliquer la règle :

$$xframe(sub, m, pc, stack(b(stack(a, s))), l) \rightarrow xframe(xsub(a, b), m, pc, s, l).$$

Une fois que le résultat est calculé grâce à toutes les règles de réécriture de $xsub$ (voir figure 6.2), il est placé en sommet de pile et nous pouvons alors calculer la prochaine opération de la méthode m, *i.e.* aller au point de programme suivant en appliquant :

$$\begin{array}{rcl} xframe(result(x), m, pc, s, l) & \rightarrow & frame(m, next(pc), stack(x, s), l), \\ next(pp1) & \rightarrow & pp2. \end{array}$$

◀

Ce même principe de basculement vers une $xframe$ est appliqué pour chaque instruction.

EXEMPLE *6.2*
Supposons qu'au point de programme suivant de l'exemple 6.1 se trouve l'instruction `pop`, qui consiste à enlever le premier élément de la pile. Alors la $frame$ va basculer en $xframe$ pour pouvoir exécuter l'instruction, *i.e.* appliquer la règle :

$$frame(name(foo, A), pp2, s, l) \rightarrow xframe(pop, name(foo, A), pp2, s, l).$$

L'instruction `pop` est ensuite exécutée (*i.e.* on enlève le premier élément de la pile) et on passe au point de programme suivant :

$$\begin{array}{rcl} xframe(pop, m, pc, stack(x, s'), l) & \rightarrow & frame(m, next(pc), s', l), \\ next(pp2) & \rightarrow & pp3. \end{array}$$

◀

Étant donné que les entiers de Peano sont utilisés dans Copster pour représenter les entiers, l'évaluation de l'instruction "`if`" requiert plusieurs règles de réécriture. Par exemple, l'instruction "`if a=b then go to the program point x`" est encodé par le terme $ifEqint(x, a, b)$, et les règles de la figure 6.3 seront appliquées.

Si au point de programme pc de la méthode m nous avons un "`if`" dont la condition est une égalité entre deux éléments, alors on passe à une $xframe$ possédant l'opération correspondante (un "`if`" avec une contrainte d'égalité entre les deux premiers éléments de la pile), et allant à un point de programme x si la condition est vérifiée. Alors nous pouvons appliquer la règle

$$\begin{array}{rl} & xframe(ifACmpEq(x), m, pc, stack(b, stack(a, s)), l) \\ \rightarrow & xframe(ifEqint(x, a, b), m, pc, s, l) \end{array}$$

```
ifEqint(x,zero,zero)          →  ifXx(valtrue,x)
ifEqint(x,succ(a),pred(b))    →  ifXx(valfalse,x)
ifEqint(x,pred(a),succ(b))    →  ifXx(valfalse,x)
ifEqint(x,succ(a),succ(b))    →  ifEqint(x,a,b)
ifEqint(x,pred(a),pred(b))    →  ifEqint(x,a,b)
ifEqint(x,succ(a),zero)       →  ifXx(valfalse,x)
ifEqint(x,pred(a),zero)       →  ifXx(valfalse,x)
ifEqint(x,zero,succ(b))       →  ifXx(valfalse,x)
ifEqint(x,zero,pred(b))       →  ifXx(valfalse,x)
```

FIGURE 6.3 — Règles de réécriture dans Copster pour l'instruction "`if-then-else`" comparant l'égalité entre deux éléments.

qui va permettre de calculer la solution, en appliquant les règles $ifEqint$ détaillées en figure 6.3.

Selon le résultat retourné par ces règles, le programme va se rendre au point de programme x si la condition est vraie ou au point de programme suivant sinon. Ceci est modélisé, dans Copster, par les deux règles suivantes :

$$\begin{array}{rcl} xframe(ifXx(valtrue, x), m, pc, s, l) & \rightarrow & frame(m, x, s, l) \text{ et} \\ xframe(ifXx(valfalse, x), m, pc, s, l) & \rightarrow & frame(m, next(pc), s, l). \end{array}$$

Nous avons vu ici une modélisation de l'instruction `if-then-else` pour la condition d'égalité entre deux entiers (*i.e.* $a = b$). Il y a également des règles de réécriture permettant de modéliser cette instruction pour chaque type de condition du `if` : $ifCmpGe$ et $ifGeint$ pour $a \geq b$, $ifCmpGt$ et $ifGtint$ pour $a > b$, $ifCmpLe$ et $ifLeint$ pour $a \leq b$, $ifCmpLt$ et $ifLtint$ pour $a < b$,etc.

De nombreuses autres instructions du bytecode $Java$ se situant au niveau des $frames$ sont décrites grâce à Copster sous forme de règles de réécriture, comme par exemple les instructions `store`, `goto`, `push`, `pop`, les opérations sur les chaines de caractères telles que la concaténation de deux chaines de caractères, le retour du caractère à la i^{me} position, etc.

Les instructions se situant au niveau de l'état du programme se font par réécriture du terme $state(f, fs, h, k)$, et concernent par exemple les invocations dynamiques de méthodes.

En effet, la règle :

$$frame(name(foo, A), pp3, s, l) \rightarrow xframe(invokeVirtual(set), name(foo, A), pp3, s, l)$$

signifie que l'instruction d'invocation dynamique de méthode est appelée dans la méthode foo de la classe A. Ici, c'est la méthode set de la classe A qui est invoquée. L'instruction d'invocation (ou d'appel) s'exécute ensuite au niveau de l'état du programme :

$$\begin{array}{lcl} state(xframe(invokeVirtual(set), m, pc, & \rightarrow & state(frame(name(set, A), pp0, s, \\ stack(loc(A, adr), stack(x, s)), l), & & locals(loc(A, adr), x, nillocal)), \\ fs, h, k) & & stack(storedframe(m, pc, s, l), fs), h, k) \end{array}$$

qui permet de créer la $frame$ permettant au programme d'exécuter la méthode invoquée, *i.e.* la méthode set de la classe A. Les règles de réécriture implémentant ce type d'instructions sont détaillées dans [Boichut et al., 2007].

Intégration du *multithread*.

La gestion du *multithread* $Java$ est intégrée dans Copster grâce à quelques règles de réécriture. Un programme est souvent conçu comme une suite (séquentielle) d'instructions. Néanmoins, il est possible de concevoir des programmes où plusieurs tâches se déroulent simultanément, en parallèle. Ces différentes tâches portent le nom de *threads* et on dit alors que l'application est *multithread*. Les threads partagent une mémoire commune et peuvent être utilisés pour exécuter plus rapidement une même portion de code.

Cependant, pour éviter de quelconques conflits d'accès à la mémoire partagée, et éviter que les données manipulées ne soient altérées, il est possible de restreindre l'accès à certaines zones mémoires à un seul thread à la fois, *i.e.* réserver l'accès en lecture et en écriture d'un objet à un seul thread. Ceci est effectué en prenant un verrou (ou ***lock***) sur l'objet réservé. Le lock est implémenté en $Java$ par la méthode `synchronize(x)`, où `x` est l'objet réservé. Cette méthode est suivie de la portion de code dans laquelle l'objet `x` ne peut être modifié que par un seul *thread* à la fois.

Dans le bytecode $Java$, cette méthode est modélisée par les instructions :

- `monitorenter` qui dénote l'entrée en section synchronisée et prend un *lock* sur l'objet en haut de pile de la *frame*, et
- `monitorexit` pour notifier la fin de la section synchronisée et rendre le *lock*.

Le *scheduler* (ou ordonnanceur) de *thread* va permettre de répartir les ressources le plus équitablement possible entre les différents *threads*.

Les *threads* sont également munis d'opérations de contrôle depuis l'extérieur. En voici quelques exemples :

- L'opération `t.join()` permet de bloquer le *thread* `t` jusqu'à l'arrêt d'un autre *thread*.
- L'opération `o.wait()`, utilisée par un objet `o` dans la méthode `synchronize(o)` (pour le même objet) permet à l'objet `o` de mettre en attente le *thread* qui exécute l'instruction `wait`. Il est important de noter que lorsque la méthode `wait` suspend le *thread* en cours d'exécution, le *lock* sur l'objet `o` est rendu.
- L'opération `o.notify()` permet de relancer un *thread* suspendu par l'opération `o.wait()`.
- etc.

Dans Copster, un *thread* est modélisé par un terme :

$$thread(loc(c,a), frame(m,pc,s,l), callstack(storedframe(m',pc',s',l'),f))$$

où :

1. $loc(c,a)$ indique la classe du $thread$ (c) et sa référence (a),
2. $frame(m,pc,s,l)$ indique la $frame$ où le *thread* possède un *lock*.
3. $callstack(storedframe(m',pc',s',l'),f)$ indique la pile d'appel du *thread*, *i.e.* la pile des *frames* que doit exécuter le *thread*.

Étant donné que toutes les instructions sont exécutées par les différents *threads*, un état du programme $Java$ est désormais modélisé par un terme :

$$state(threadlist(thread(loc,f,cs), threadlist(t,tl)), h, k, tt)$$

où :

1. $state(threadlist(thread(loc, f, cs), threadlist(t, tl))$ est la liste des *threads* du programme,
2. h est le tas des objets non locaux comme dans le cas du *monothread* (voir paragraphe **Formalisation des états et des *frames***),
3. k est le tas des champs statiques comme dans le cas du *monothread*,
4. $tt : Objet \rightarrow Objet \times Entier$ est une table associant à chaque objet locké l'objet *thread* le détenant et combien de fois.

Si l'on compare avec un état de programme dans le cas du *monothread* (soit le terme $state(f, fs, h, s)$), on peut noter que la frame courante f et la liste de frame fs disparaissent en faveur de la liste de *threads*. En effet, les frames courantes sont désormais contenues dans les *threads* qui vont les exécuter. On note aussi l'apparition d'une table lt associant les objets lockés aux *threads* possédant le *lock*.

Présentons maintenant quelques modélisations d'instructions sur les *threads* en règles de réécriture. Une modélisation simple est celle du *scheduler* basique, qui va permettre de basculer entre les différents threads :

$$threadlist(x, threadlist(y, z)) \rightarrow threadlist(y, threadlist(x, z)).$$

Ici, le *scheduler* permet de basculer du *thread* x, situé en haut de la pile, au *thread* y, en le plaçant en haut de la pile. L'instruction `monitorenter` est modélisée de cette manière :

$$\begin{array}{ll} & \textbf{state(}threadlist(thread(loc(c, a), xframe(\\ & monitorenter, m, pc, stack(x, s), l), cs), tl)\textbf{,}h\textbf{,k,tt)} \\ \rightarrow & \textbf{state(}threadlist(thread(loc(c, a), xframe(\\ & xMonitorEnter(xGetLock(a, xGetHeapStack(x,h))), m, pc, stack(x, s), l), cs), tl)\textbf{,h,k,tt)} \end{array}$$

La partie gauche de la règle modélise le fait que le premier *thread* de la liste est localisé à l'adresse a de la classe c, et possède un *lock* sur une frame exécutant l'instruction $monitorenter$ (notifié par le terme $xframe(monitorenter, m, pc, stack(x, s)$ en deuxième paramètre du *thread*).

cs est la pile d'appel du premier *thread*, tl le reste de la liste de thread, h, k, tt les paramètres connus d'un état (**state**) de programme expliqués au-dessus. Dans la partie droite, on remarque que l'instruction $monitorenter$ lance l'instruction $xMonitorEnter$ qui va permettre de prendre un *lock* sur l'objet x en haut de la pile grâce à l'instruction $xGetLock(a, xGetHeapStack(x,h))$, avec a l'adresse du *thread*, pour pouvoir lui associer l'objet x, et $xGetHeapStack(x,h)$ permettant de récupérer les informations associées à l'objet x dans le tas h des objets globaux.

L'instruction $xGetLock$ va par la suite permettre de stocker $(x,a, n + 1)$ dans la table tt, avec n le nombre de fois où x était déjà locké par a.

Les autres fonctions de synchronisations ainsi que les fonctions de contrôle telles que `join`, `wait`, et `notify`, sont également modélisées dans Copster.

6.2.2 Implémentation et exemple

Soit P_J un programme $Java$, que l'on veut transformer en un ensemble de règles de réécriture grâce à Copster.

Copster est composé :

- de fichiers `.rex` d'une certaine syntaxe, représentant un ensemble de règles de réécriture à générer selon certains paramètres,
- d'un parseur en `Ocaml` capable d'interpréter les fichiers `.rex` en leur donnant les bons paramètres, qu'il va récupérer depuis le bytecode $Java$ du programme P_J à transformer, afin de générer les règles de réécriture correspondant à P_J.

Les règles de réécriture sont tout d'abord générées en un format appelé `aterm`, qui va ensuite permettre de générer des fichiers de règles de réécriture au format Timbuk, mais aussi au format `Maude`.

Les fichiers `.rex` sont composés de plusieurs mots clés et fonctions qui vont être interprétés par le parseur pour la génération de règles. Ainsi, dans un fichier `.rex`, la fonction `genrule(l,r)` permet de générer une règle de réécriture $l \to r$, et la fonction `var(x)` va permettre de donner un nom disponible à une variable de la règle pour éviter les conflits.

EXEMPLE *6.3*
La ligne : `genrule(xadd(pred(var(a)),succ(var(b))),xadd(var(a),var(b)));` d'un fichier `.rex` va permettre de générer la règle de réécriture :

$$xadd(pred(a), succ(b)) \to xadd(a, b))$$

◀

D'autres intructions plus complexes vont être également interprétées par le parseur, afin par exemple de générer l'ensemble des points de programmes de P_J, ou l'ensemble des instructions bytecode utilisées par P_J, selon le point de programme où elles se situent, dans quelle classe, et dans quelle méthode. C'est ce que nous pouvons voir sur l'exemple 6.4 ci-dessous permettant de générer le premier pas pour toutes les instructions du programme (*i.e.* le basculement de $frame$ en $xframe$ comme expliqué dans la sous-section 6.2.1).

EXEMPLE *6.4*
Les variables dépendant du programme $Java$ à transformer (P_J) sont précédées du symbole $.

```
(* Génération de la première étape pour *)
(*toutes les instructions du programme *)
for i from 1 to $nb_insts
(* Pour toutes les instructions *)
do(
  let inst = getn($insts,$i)
  (* On récupère l'instruction i dans la variable inst *)
  in (
    let class_name = getn($inst,1)
    (* On récupère le nom de la classe où se situe l'instruction inst *)
    and method_name = getn($inst,2)
```

```
    (* Ainsi que le nom de la méthode où se situe l'instruction inst *)
    and pc = getn($inst,3)
    (* Et le point de programme où se situe l'instruction inst *)
    and opcode = getn($inst,4)
    (* Et enfin le nom  de l'instruction inst *)
    in (
      genrule(
          frame(name($method_name,$class_name),$pc,var(s),var(l)),
          xframe($opcode,name($method_name,$class_name),$pc,var(s),var(l))
      );
      (* Puis on peut générer la règle de réécriture correspondante *)
    );
  );
);
```

Si au point de programme `pp1` du bytecode de P_J se trouve l'instruction `pop`, qui se situe dans la méthode `m5` de la classe `A`, et si au point de programme `pp18` se trouve l'instruction `add`, se situant dans la méthode `m2` de la classe `B`, alors les deux règles suivantes :

$$\begin{array}{rcl} frame(name(m5, A), pp1, s, l) & \rightarrow & xframe(pop, name(m5, A), pp1, s, l) \\ frame(name(m2, B), pp18, s, l) & \rightarrow & xframe(add, name(m2, B), pp18, s, l), \end{array}$$

seront générées par le parseur au moment de l'interprétation des lignes détaillées au début de cet exemple, ainsi que les règles correspondant aux instructions de tous les points de programmes de P_J. ◀

Le parseur fonctionne schématiquement de la manière suivante :

(1). Il parse :

  (A). le programme $Java$ donné en entrée afin de récupérer toutes les informations nécessaires (*i.e.* nombre de points de programmes, nombre et noms des classes, noms des méthodes, les instructions utilisées, etc.)

  (B). les fichiers `.rex`, en utilisant les informations récupérées en parsant le programme $Java$.

(2). Exporte les données parsées au bon format (Timbuk ou `Maude`).

Nous allons désormais détailler ce fonctionnement en décrivant les fichiers `.ml` principaux. Le fonctionnement général est également représenté en figure 6.5.

(1). (A). – Le fichier `JClassAbs.ml` fournit une réprésentation du programme $Java$. Il parse le programme $Java$ passé en entrée afin de le transformer dans un type `program`, qui contient une liste de classe, le nombre de points de programme maximum et le nombre de variables locales maximum.

```
type program = {
  p_structure : interface_or_class list;
  p_max_pcs : int;
  p_max_locals : int
}
```

- Le fichier `gen.ml` contient les fonctions nécessaires pour générer, à partir d'un `program`, toutes les informations utiles du programme *Java*, comme le nombre de méthodes (`get_nb_methods`), de classes (`get_nb_classes`), le nombre de champs par méthodes (`get_nb_fields_per_classes`), etc.
- Le fichier `import.ml` va utiliser le `program` renvoyé par `JClassAbs.ml` et les fonctions de `gen.ml` pour renvoyer une liste de `string` utilisable par le parseur des fichiers `.rex` détaillé plus loin ((B).), contenant toutes les informations nécessaires du programme *Java* :

```
(nb insts = 29, max locals = 3,
max pc = 8, nb classes = 12,
[...],
nb fields per classes
= (0,1,1,0,0,0,0,2,0,0,0,0),
[...])
```

(B). – Le fichier `syntax_tree.ml` contient le type `insts` permettant de représenter les instructions des fichiers `.rex`, *i.e.* tout ce qui ne concerne pas les règles de réécriture en elle-même, mais ce qui doit être parsé afin de les générer. Comme nous pouvons le voir dans l'exemple 6.4, c'est ce qui permet par exemple de générer une règle de réécriture (`genrule`), ou faire une boucle `for`, ou récupérer une donnée du programme *Java* retournée par `import.ml` (`getn`).

```
type insts =
  | Inst of inst * insts * string * int
  | Nop
and inst =
  | Genrule of expr * expr
  | Set of string * expr
  | Setn of string * expr * expr
  | If of expr * insts * insts
  | For of string * range * insts
  | Expr of expr
  [...]
and expr = simple_expr * string * int
  | Depth of expr
  | Not of expr
  | And of expr list
  | Or of expr list
  | Equal of expr * expr
  | Sup of expr * expr
  [...]
```

- Le fichier `environment.ml` fournit le type `environment`, qui permet de stocker et de modifier les données, une fois parsées par l'*interpréteur* détaillé ci-après, pour pouvoir exporter un fichier au format Timbuk. Il permet de fournir la liste des symboles fonctionnels et la liste des variables utilisés par les règles de réécriture, et bien sur la liste des règles de réécritures interprétées, représentant le comportement du programme *Java* passé en entrée. Il fournit également les opérations nécessaires pour ajouter les éléments

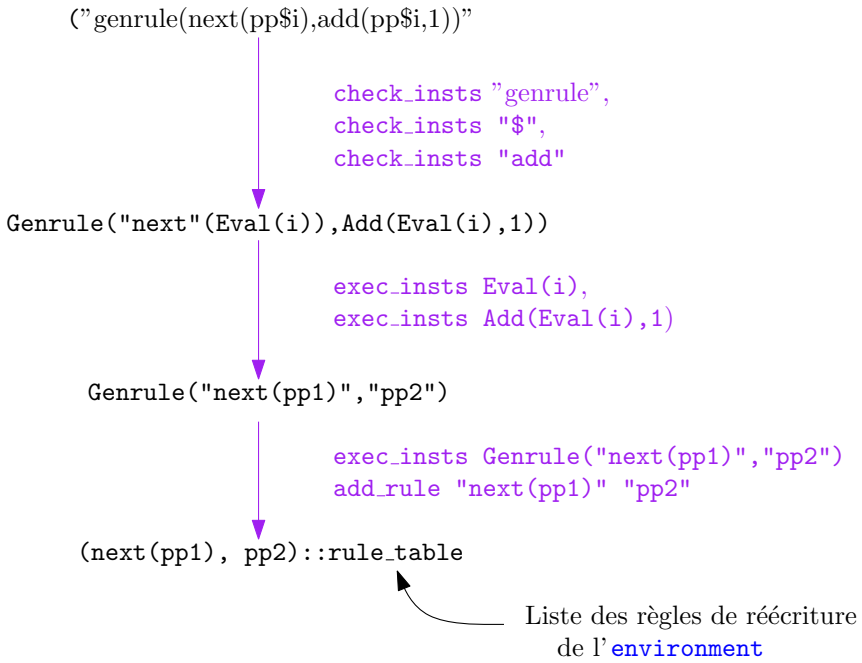


FIGURE 6.4 — Fonctionnement de l'*interpréteur* (avec variable `i` évaluée à 1).

dans un `environment`, comme par exemple la fonction `add_rule l r` qui permet d'ajouter une règle de réécriture $l \to r$ dans la liste des règles de réécriture (appelée `rule_table`).

– L'*interpréteur*, *i.e.* le fichier `interpreter.ml` :

  i. Transforme les différents éléments des fichiers `.rex` en éléments de type `insts` du fichier `syntax_tree.ml` : grâce à la fonction `check_insts`.

  ii. Parse les `insts` afin de générer les règles de réécritures correspondantes au programme $Java$ passé en entrée et ainsi ajouter à l'`environment` les éléments nécessaires : grâce à la fonction `exec_inst`.

  Par exemple, le *parsing* de l'`insts` `Getn(i,n)` va récupérer l'information nécessaire importée par le fichier `import.ml`, et le parsing de l'instruction `Genrule(l,r)` va permettre d'ajouter le tuple `(l,r)` dans la liste des règles de réécritures de l'`environment`, en appelant la fonction `add_rule l r`. Un exemple de son comportement est représenté en figure 6.4.

(2). – Le fichier `export.ml`, génère tout d'abord, depuis l'`environment` calculé par l'*interpréteur*, un fichier dans un format spécial appelé `aterm`.

– Ce fichier `.aterm` peut ensuite être interprété par un parseur contenu dans le fichier `atermParser.ml`, qui va permettre de le transformer en un élément de type `specification` qui sera utilisable pour générer les fichiers au format Timbuk ou `Maude`.

– Grâce à la specification calculée par `atermParser.ml`, `export.ml` génère les fichiers Timbuk et/ou `Maude` représentant le comportement du programme $Java$ passé en entrée.

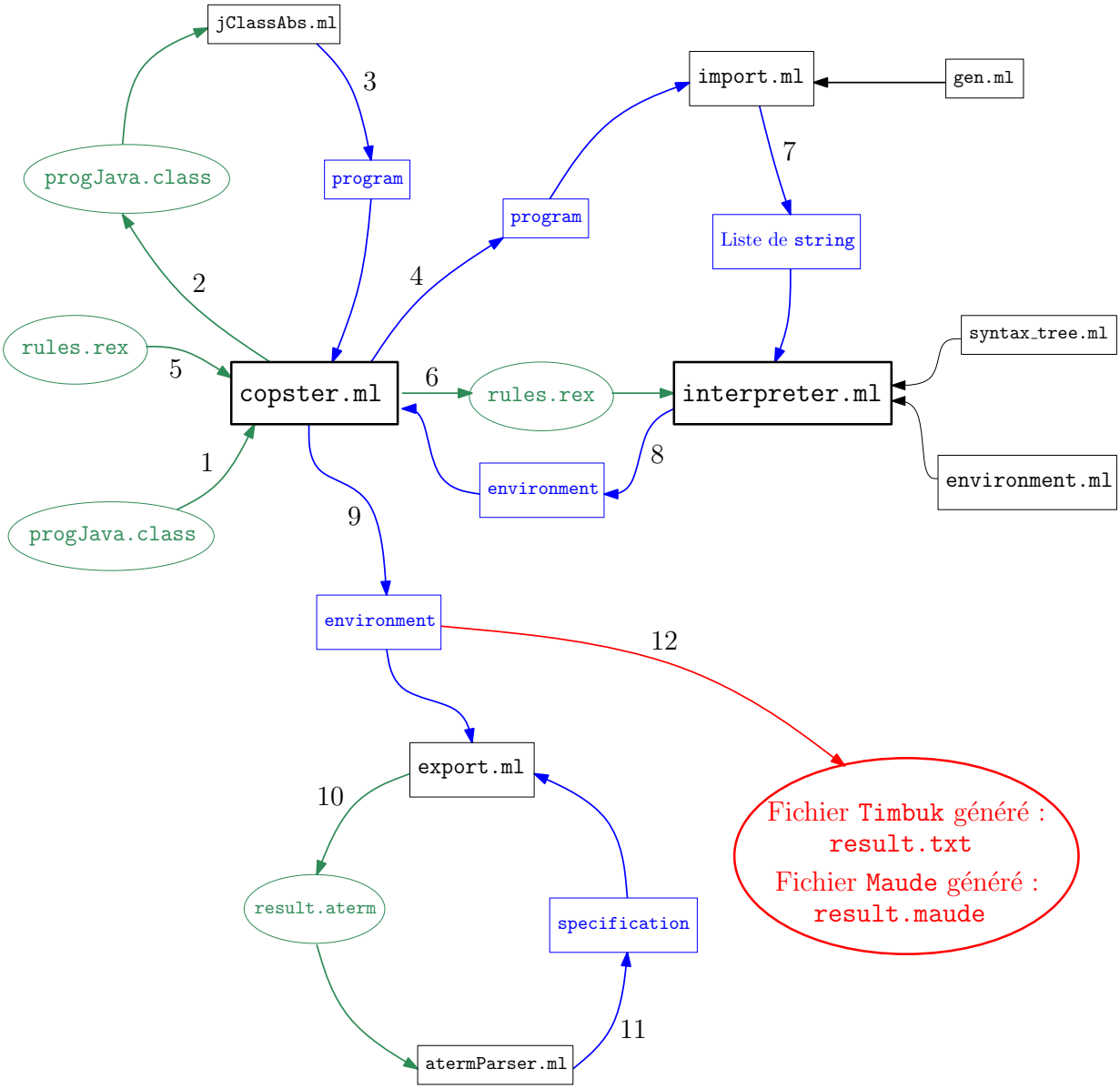


FIGURE 6.5 — Fonctionnement du parseur de Copster.

Puis enfin, le fichier principal `copster.ml` va appeler toutes les fonctions nécessaires des fichiers décrits ci-dessus afin de générer le fichier Timbuk correspondant au programme *Java* que l'utilisateur lui donnera en paramètre.

Le comportement du parseur de Copster est résumé sur la figure 6.5 : le fichier `copster.ml` appelle la fonction `full_parse` de la classe `jClassAbs` en lui passant le programme *Java* en paramètre (1 et 2), puis récupère le `program` généré par cette fonction (3). Ensuite, `copster.ml` donne le `program` généré au fichier `import.ml` (4). Puis `copster.ml` appelle la fonction `parse_program` du fichier `interpreter.ml` en lui donnant le fichier `rules.rex` en paramètre (5 et 6). Grâce au fichier `rules.rex` récupéré et aux informations sur le programme *Java* calculées par le fichier `import.ml` (7), l'interpréteur va généré un `environment` contenant les règles de réécriture correspondant au programme *Java* (8). Le fichier `copster.ml` va ensuite appeler le fichier `export.ml` en lui donnant l'`environment` (9), et va ainsi permettre au fichier `export.ml` de générer

```
public class Factorielle {
    public static int factorial(int i){
        int res=1;
        for (int j=i; j>1; j--){
            res=res*j;
        }
        return res;
    }

    public static void main(String arg[]){
        int x=0;
        try {x=System.in.read();} catch(java.io.IOException e){};
        if (x>=0) System.out.println(factorial(x));
    }
}
```

FIGURE 6.6 — Programme `.java` calculant la factorielle d'un nombre passé en entrée.

un fichier `.aterm` (10) qui sera parsé et redonné à `export.ml` sous forme de specification (11) pour lui permettre de générer les fichiers Timbuk ou `Maude` représentant le comportement du programme $Java$.

Donnons maintenant un exemple de programme $Java$ et sa transformation en règles de réécriture au format Timbuk. Le programme $Java$ de la figure 6.6 prend un entier en entrée, puis affiche la factorielle de cet entier à l'écran.

Tout d'abord, il faut compiler ce fichier `.java` en fichier `.class`, afin de pouvoir obtenir le bytecode de ce programme. Supposons que le fichier `.class` se situe dans un dossier `/test` dans le dossier `/copster`, il suffit d'entrer la commande suivante dans le dossier `/copster` :

```
./build/copster -rules ./rules/monothread/rules.rex -stubs
./rules/stubclasses.jstub -classpath./tests/ -javaclass Factorielle
-aterms Factorielle.aterms -timbuk Factorielle.txt
```

Cette commande va compiler, dans le cas *monothread*, le fichier `aterm` et le fichier Timbuk correspondant au programme de la figure 6.6.

Nous allons maintenant décrire certains extraits du fichier Timbuk résultat. Ce fichier comporte 638 règles de réécriture, dont environ 50 dépendantes du programme à modéliser et 588 fixes (indépendantes du programme).

Tout d'abord se trouvent tous les symboles fonctionnels `Ops` modélisant les instructions ou les structures du programme.

```
Ops outstack:2 pp14:0 Factorielle:0 IOException:0 Thread:0 InterruptedException:0
    in:0 out:0 Field:3 factorial:0 println:0 read:0 AccDefault:0 join:0 start:0
   [...]
   isInit:1 AccNative:0 stack:2 loc:2 typeInstance:3 localCall:3 rmArgs:3 valtrue:0
   accessibleMC:2 lookup:2 lookupIV:2 lookupSp1:2 init:0 clinit:0 AccStatic:0
```

Ensuite sont listées les différentes variables `Vars` des règles de réécriture :

```
Vars l0 l1 l2 [...] sl ca cam cc ic rc fx
```

Puis vient l'ensemble du système de réécriture S représentant la sémantique du bytecode associé au programme de la figure 6.6, commençant par la description de l'état initial du programme (`initialJavaState`). Le symbole `instackExample` est une liste permettant de modéliser les éléments passés en entrée du programme. Ici, les entiers saisis au clavier sont modélisés par la liste $(3, 1, 2, 0)$. C'est donc la factorielle de 3 (premier entier saisi au clavier) que le programme va calculer puis afficher.

```
TRS S
initialJavaState(x) ->
   IO(state(frame(name(Method(main,ConsType(AType(OType(Class(ConsName(java,ConsName(
   lang,ConsName(String,NilName)))))),NilType),void),Class(ConsName(Factorielle,
   NilName))),pp0,nilstack,locals(nillocal,nillocal,nillocal)),nilcallstack,heaps(
   heap(zero,nilstack0),heap(succ(zero),stack1(object1,nilstack1)),heap(zero,
   nilstack2),heap(succ(zero),stack3(object3,nilstack3)),heap(zero,nilstack4),heap(
   zero,nilstack5),heap(zero,nilstack6),heap(zero,nilstack7),heap(zero,nilstack8),
   heap(zero,nilstack9)),heaps(object0,object1,object2,object3,object4(loc(Class(
   ConsName(java,ConsName(io,ConsName(InputStream,NilName)))),zero),loc(Class(
   ConsName(java,ConsName(io,ConsName(PrintStream,NilName)))),zero)),object5,
   object6,object7,object8,object9)),x,niloutstack)
instackExample ->
   instack(succ(succ(succ(zero))),instack(succ(zero),instack(succ(succ(zero)),instack(
   zero,instack(pred(zero),nilinstack)))))
```

Vient ensuite un exemple de règle de réécriture se situant au niveau de l'état du programme.

```
state(frame(name(Method(init,NilType,void),Class(ConsName(java,ConsName(lang,ConsName(
   Object,NilName))))),pp0,nilstack,ll),callstack(storedframe(m,pc,s,l),f),h,k) ->
   state(frame(m,next(pc),s,l),f,h,k)
```

Puis les règles modélisant les opérations arithmétiques (telles que `xadd`, `xsub`, etc.), ainsi que les calculs booléens des conditions de l'intruction `if-then-else`.

```
xframe(result(x),m,pc,s,l) -> frame(m,next(pc),stack(x,s),l)
xadd(zero,zero) -> result(zero)
[...]
ifLtint(x,zero,succ(b)) -> ifXx(valtrue,x)
ifLtint(x,zero,pred(b)) -> ifXx(valfalse,x)
```

Voici ensuite quelques règles `frame` $\rightarrow$ `xframe` pour la fonction `factorial` de la classe `Factorielle`. On voit ici par exemple un `push(1)` au point de programme `pp0` de la méthode `factorial`. Soit `a` et `b` les deux premiers éléments de la pile, on voit ensuite un `ifCmpLe(pp13)` au point de programme `pp6`, qui représente l'instruction `if a=<b then go to the program point 13`.

```
[...]
frame(name(Method(factorial,ConsType(TInt,NilType),TInt),Class(ConsName(Factorielle,
   NilName))),pp0,s,l) ->
   xframe(push(succ(zero)),name(Method(factorial,ConsType(TInt,NilType),TInt),Class(
   ConsName(Factorielle,NilName))),pp0,s,l)
[...]
frame(name(Method(factorial,ConsType(TInt,NilType),TInt),Class(ConsName(Factorielle,
   NilName))),pp6,s,l) ->
   xframe(ifCmpLe(pp13),name(Method(factorial,ConsType(TInt,NilType),TInt),Class(
   ConsName(Factorielle,NilName))),pp6,s,l)
[...]
```

Puis l'ensemble des règles de réécriture permettant de passer d'un point de programme au suivant.

```
[...]
next(pp0) -> pp1
next(pp1) -> pp2
[...]
next(pp12) -> pp13
next(pp13) -> pp14
[...]
```

Puis ensuite quelques modélisation d'instructions telles que `pop`, `load`, ou `store`.

```
[...]
xframe(pop,m,pc,stack(x,s),l) -> frame(m,next(pc),s,l)
[...]
xframe(load(local0),m,pc,s,locals(l0,l1,l2)) ->
   frame(m,next(pc),stack(l0,s),locals(l0,l1,l2))
xframe(load(local1),m,pc,s,locals(l0,l1,l2)) ->
   frame(m,next(pc),stack(l1,s),locals(l0,l1,l2))
[...]
xframe(store(local0),m,pc,stack(x,s),locals(l0,l1,l2))
-> frame(m,next(pc),s,locals(x,l1,l2))
[...]
```

Et enfin, le terme initial `Init` sur lequel l'ensemble des règles de réécriture sera appliqué.

```
Set Init
initialJavaState(instackExample)
```

De nombreuses règles de réécriture sont bien entendue occultées dans cet exemple.

6.3 Implémentation de CopsterLTA et TimbukLTA

Nous avons développé une version de Copster [Barré et al., 2009] (appelée CopsterLTA), qui est capable de compiler un fichier *Java* `.class` en système de réécriture *conditionnel* avec termes *interprétables* comme nous l'avons vu au chapitre précédent (*i.e.* termes contenant les éléments d'un domaine infini tels que les entiers, ainsi que ces opérations qui sont évaluées sans passer par l'application de règles de réécriture). Nous avons également développé TimbukLTA, une version de Timbuk gérant la complétion LTA décrite au chapitre précédent. Pour l'instant, ces implémentations gèrent le domaine des entiers relatifs et les intervalles sur les entiers comme domaine abstrait correspondant. TimbukLTA et CopsterLTA sont disponibles sur les pages http://www.irisa.fr/celtique/genet/timbuk/ et http://www.irisa.fr/celtique/genet/copster/.

6.3.1 CopsterLTA

Comme nous l'avons déjà un peu expliqué durant l'introduction, l'intérêt majeur de notre technique est la disparition totale de la modélisation des opérations arithmétiques par application successive de plusieurs règles de réécriture.

Dans le Copster d'origine, la règle :

$$xframe(sub, m, pc, stack(b(stack(a, s))), l) \rightarrow xframe(xsub(a, b), m, pc, s, l)$$

est utilisée pour passer au calcul de la soustraction entre les deux premiers éléments de la pile. Elle permet donc d'appeler le terme $xsub(a, b)$ afin d'appliquer plusieurs fois les sept règles de la soustractions vues en figure 6.2, jusqu'à obtenir le résultat res de la soustraction dans le terme $result(res)$.

Dans CopsterLTA, la règle permettant de passer de sub à $xsub$ décrite au-dessus, ainsi que les sept règles de la soustraction sont remplacées par une seule règle, appliquée une seule fois :

$$xframe(sub, m, pc, stack(b(stack(a, s))), l) \rightarrow xframe(result(a - b), m, pc, s, l).$$

On voit ici que l'opération $a-b$ à évaluer est directement insérée dans le terme résultat. Ainsi, l'opération $300 - 400$ qui nécessitait 300 applications des règles de réécriture dans Copster, ne nécessite désormais que l'application d'une seule règle, et d'une étape d'évaluation de $a - b$.

Il en est de même pour l'addition, ainsi que pour la multiplication :

$$(xframe(mult, m, pc, stack(b, stack(a, s)), l) \rightarrow xframe(result(a \times b), m, pc, s, l)).$$

La multiplication $a \times b$ qui nécessitait $a \times (b+2)$ applications de règles de réécriture dans Copster (*i.e.* 850 applications de règles pour l'opération 50×15), ne nécessite également plus qu'une seule application, et une étape d'évaluation.

En outre, d'autres opérations telles que le `if-then-else` peuvent être simplifiées en utilisant notre formalisme. Toutes les règles de ce type, permettant d'implémenter les différentes conditions de l'instruction `if-then-else` sur les entiers (*i.e.* $a = b$, $a < b$, etc., avec a et b des entiers relatifs) vont disparaitre avec l'utilisation des LTA car la comparaison entre deux éléments peut être directement évaluée. En effet, par exemple, l'égalité entre deux entiers fait partie des prédicats prédéfinis que l'on peut utiliser dans les contraintes des règles de réécriture. Rappelons toutes les règles permettant d'exécuter l'instruction "if-then-else" dans le cas d'une comparaison d'*égalité* entre deux éléments. Il y a tout d'abord les règles décrites dans la figure 6.3, ainsi que les trois règles décrites en section précédente :

$$\begin{array}{rl} & xframe(ifACmpEq(x), m, pc, stack(b, stack(a, s)), l) \\ \rightarrow & xframe(ifEqint(x, a, b), m, pc, s, l), \\ xframe(ifXx(valtrue, x), m, pc, s, l) & \rightarrow \quad frame(m, x, s, l) \text{ et} \\ xframe(ifXx(valfalse, x), m, pc, s, l) & \rightarrow \quad frame(m, next(pc), s, l. \end{array}$$

(*i.e.* 12 règles au total). Toutes ses règles de réécriture seront remplacées dans CopsterLTA par les deux règles de réécritures **conditionnelles** suivantes :

$$xframe(ifACmpEq(x), m, pc, stack(b, stack(a, s)), l) \rightarrow frame(m, x, s, l) \Leftarrow a = b$$
(si $a = b$ nous allons au point de programme p), et
$$xframe(ifACmpEq(x), m, pc, stack(b, stack(a, s)), l) \rightarrow frame(m, x, s, l) \Leftarrow a \neq b$$
(si $a \neq b$ nous allons au point de programme suivant).

Remarque. En pratique, la condition $a \neq b$ est remplacée par la condition $a < b \wedge a > b$ pour un meilleur filtrage de la solution retournée, ce qui sera détaillé en sous-section 6.3.2 suivante.

Il en est de même pour toutes les autres règles de réécriture décrivant l'instruction `if-then-else` selon sa condition de test (`ifCmpGe` et `ifGeint` pour $a \geq b$, `ifCmpGt` et `ifGtint` pour $a > b$, `ifCmpLe` et `ifLeint` pour $a \leq b$, `ifCmpLt` et `ifLtint` pour $a < b$, etc.).

D'autres modifications ont dû être effectuées, pour remplacer des entiers de Peano par des entiers, mais ces quelques modifications n'apportent pas de gain de temps ou de règles appliquées. Par exemple, les objets sont référencés par des entiers et non pas des entiers de Peano, les fonctions sur les chaines de caractères comme par exemple récupérer le n^{eme} caractère ont également été modifiées :

$$\text{Dans Copster :} \quad \begin{array}{rcl} nth(zero, Cons(c,n)) & \rightarrow & resultchar(c) \\ nth(succ(i), Cons(c,n)) & \rightarrow & nth(i,n) \end{array}$$

$$\text{Dans CopsterLTA :} \quad \begin{array}{rcl} nth(0, Cons(c,n)) & \rightarrow & resultchar(c) \\ nth(i, Cons(c,n)) & \rightarrow & nth(minus(i,1),n) \Leftarrow (i > 0) \end{array}$$

Les fichiers de règles `.rex` ont été modifiés, aussi bien dans le cas *monothread* que dans le cas *multithread*. La gestion des *threads* a nécessité quelques transformations, notamment l'utilisation de règles conditionnelles.

Le parseur a été également modifié pour prendre en compte les nouveaux symboles ainsi que les règles conditionnelles. Ainsi, des instructions (type `insts`) ont été rajoutées pour les opérations arithmétiques (+,-,×,etc.), et pour les prédicats des règles conditionnelles ($<, >$, etc.). En voici un exemple avec l'opérateur "+".

$$\texttt{"plus(var(a),var(b))"} \xrightarrow{check_insts} \texttt{Plus(a,b)} \xrightarrow{exec_insts} \texttt{"a+b"}$$

Détaillons un peu : le parseur lit la chaine "`plus(a,b)`", contenue dans la règle suivante du fichier `rules.rex`.

```
genrule(
  xframe(add,var(m),var(pc),stack(var(b),stack(var(a),var(s))),var(l)),
  xframe(result(plus(var(a),var(b))),var(m),var(pc),var(s),var(l))
);
```

Ensuite, la fonction `check_insts` du fichier `interpreter.ml` la transforme en terme `Plus(a,b)`. Puis ce terme est transformé en la chaine de caractère "`a+b`" qui va apparaitre dans le fichier Timbuk (ou `Maude`) généré.

L'instruction `GenruleCond` et son *parsing* ont été rajoutés pour permettre de parser les règles de réécriture conditionnelles, et le type `environment` possède une liste supplémentaire contenant l'ensemble des règles de réécriture conditionnelles (ainsi que les fonctions d'ajout associées telles que `add_rules_cond`).

$$\begin{array}{l} \texttt{"genrulecond(l,r,c)"} \xrightarrow{check_insts} \texttt{GenruleCond(l,r,c)} \\ \xrightarrow{exec_insts, add_rule_cond} \texttt{(l,r,c)::rule_cond_table} \end{array}$$

Les fichiers `syntax_tree.ml`, `environment.ml`, `interpreter.ml`, `export.ml` ainsi que `atermParser.ml` (que l'on voit en figure 6.5) ont donc été modifiés.

Afin de pouvoir fonctionner avec l'outil `Maude` (pour que les termes tels que "`a+b`" puissent être gérés), le fichier `Maude` généré appelle un fichier `opLTA.maude` que nous avons rajouté, où ces opérations sont définies.

6.3.2 TimbukLTA

Dans cette sous-section, nous allons décrire les modules `.ml` ajoutés et intégrés à Timbuk afin de pouvoir effectuer la complétion LTA dans le cas des entiers relatifs et des intervalles. Nous allons donc détailler l'implémentation des intervalles et le fonctionnement du solveur dans TimbukLTA, ainsi que d'autres opérations importantes telles que l'évaluation des opérations entre les intervalles (pour effectuer directement l'évaluation d'une opération sans appliquer de nombreuses règles de réécriture), ou l'opérateur de *widening*.

Pour une contrainte donnée sur des variables dont les valeurs sont définies par des intervalles, le solveur donne une solution approchée. Pour définir ce solveur, nous devons définir le type des intervalles, utilisant le module "`Bounds`" décrit ci-dessous.

Module `Bounds`

Un objet du type `Bounds` est soit un entier relatif, soit $+\infty$, soit $-\infty$, pour lequel nous avons défini plusieurs fonctions, comme la transformation d'un `Bound` en un entier relatif (ou renvoie une exception dans le cas d'un `Bound` infini), et d'un entier relatif en un `Bound`, ainsi que certaines opérations de comparaison comme $<, >, min, max, +, -, \ldots$

Module `Intervals`

Un objet du type `Intervals` est un couple de `Bounds` : une borne supérieure (de type `Bounds`, *i.e.* un entier ou $+\infty$ ou $-\infty$) et une borne inférieure (de type `Bounds`). Pour un intervalle i donné, $sup.i$ correspond à la borne supérieure de i et $inf.i$ correspond à la borne inférieure de i. Ce module est également fournit avec des fonctions de comparaisons sur les intervalles, ainsi que les opérations arithmétiques décrites ci-dessous :

- $[a, b] + [c, d] = [a + c, b + d]$
- $[a, b] - [c, d] = [a - d, b - c]$
- $[a, b] * [c, d] = [min(a * c, a * d, b * c, b * d), max(a * c, a * d, b * c, b * d)]$
- $[a, b]/[c, d] = [min(a/c, a/d, b/c, b/d), max(a/c, a/d, b/c, b/d)]$

L'opérateur de *widening* est également défini dans ce module et permet de retourner une abstraction étant donné deux intervalles. Par exemple, si les intervalles $[2, 3]$ et $[3, 4]$ sont passés en paramètres, alors la fonction de *widening* va retourner l'intervalle $[2, +\infty[$. Si $[0, 2]$ et $[-3, -1]$, sont passés en entrée, alors le *widening* retournera $]-\infty, 0]$.

Dans ce module sont également définies des fonctions de *restriction* qui vont être utilisées par le solveur dans un autre module. En effet, ces fonctions de restriction permettent de restreindre un intervalle selon une contrainte (contenant $=, <, \leq$ ou $\neq$) et un deuxième intervalle, pour obtenir une solution ou l'approximation d'une solution.

Voyons la syntaxe de cette fonction sur quelques exemples :

- $restrict_var_varC_leq\ i_1\ i_2$ avec i_1 et i_2 deux intervalles : un de ces deux intervalles sera restreint selon la contrainte $i_1 \leq i_2$. Ici, $varC$ représente la position de l'intervalle constant (donc non restreint). L'intervalle i_1 est donc celui qui est restreint dans ce cas.
- $restrict_varC_var_leqStr\ i_1\ i_2$: i_2 est restreint selon la contrainte $i_1 < i_2$.

Et cette fonction permet de restreindre un des intervalles passés en entrée de cette manière :

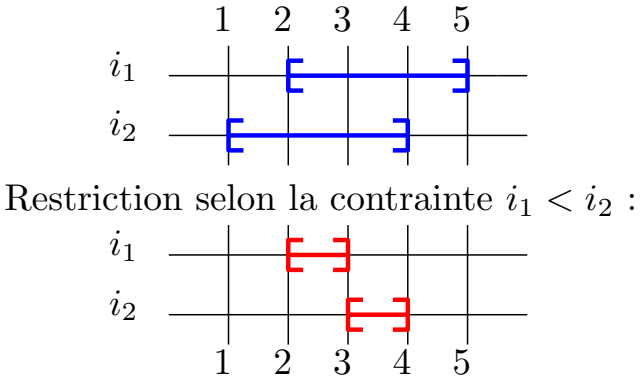


FIGURE 6.7 — Restriction des intervalles i_1 et i_2 selon la contrainte $i_1 < i_2$.

- $restrict_var_varC_leq\ i_1\ i_2$: la solution est une sur-approximation, pour éviter toute sous-approximation dans la complétion afin de préserver le théorème de complétude. Nous comparons donc ici seulement les bornes supérieures. Si $sup.i_1 \leq sup.i_2$ alors i_1 ne sera pas restreint. Sinon, $sup.i_1$ prend la valeur de $sup.i_2$. Par exemple : pour $[4, 9] \leq [5, 7]$, la restriction retournée pour i_1 est l'intervalle $[4, 7]$.
- $restrict_varC_var_leqStr\ i_1\ i_2$: cette restriction suit le même principe que la précédente, mais étant donné que c'est i_2 qui est restreint cette fois-ci, la fonction compare uniquement les bornes inférieures. Si $inf.i_1 < inf.i_2$ alors i_2 ne sera pas restreint. Sinon, $inf.i_2$ prend la valeur $inf.i_1 + 1$ (le "+1" est dû à l'inégalité stricte $i_1 < i_2$). Par exemple : pour $[3, 5] < [2, 8]$, la restriction retournée pour i_2 est $[4, 8]$.
- Différence entre deux intervalles : une règle de réécriture $l \to r \Leftarrow x \neq y$ est remplacée par deux règles de réécritures $l \to r \Leftarrow x < y$ et $l \to r \Leftarrow x > y$ utilisant les restrictions précédentes.

Un exemple de restriction est également illustré en figure 6.7 : ici, les deux intervalles sont restreints l'un après l'autre..

Module `Constraint`

Ce module contient le solveur et la fonction d'évaluation. Il définit plusieurs types :
- Le type `var` définit les variables des contraintes : un couple composé du nom de la variable et de sa valeur associée, qui est de type `Interval` ($(x, [2, 3])$ par exemple). Au niveau de la complétion, la valeur associée correspond à la substitution trouvée grâce à l'algorithme de filtrage.
- Le type `expr` définit les expressions pouvant apparaître dans une contrainte : des compositions d'entiers, de variables, et d'opérations arithmétiques. Par exemple, l'objet de type `expr` "$Add(Variable(x, [2, 3]), Mult(Entier(2), Variable(y, [3, 4])))$" modélise l'expression $x + 2 * y$ où $x = [2, 3]$ et $y = [3, 4]$.
- Le type `const` définit une contrainte : une comparaison ($=, <, >, \leq, \geq$ ou $\neq$) entre deux expressions (de type `expr`). Par exemple, $InfStr(Add(Variable(x, [2, 3]), Mult(Entier(2), Variable(y, [3, 4]))), Variable(z, [5, 6]))$ modélise la contrainte $x + 2 * y < z$.

La fonction d'évaluation, appelée `eval_expr`, retourne l'évaluation numérique d'une expression, *i.e.* l'évaluation des opérations arithmétiques sur les intervalles contenus dans les variables de l'expression. Par exemple, l'évaluation de l'expression :

$$Add(Variable(x_1, [3,5]), Mult(Variable(x_2, [1,3]), Variable(x_3, [3,4])))$$

retourne $[6, 17]$ car :
$Add(Variable(x_1, [3,5]), Mult(Variable(x_2, [1,3]), Variable(x_3, [3,4])))$
$= x_1 + x_2 * x_3 = [3,5] + [1,3] * [3,4] = [6,17]$.

La fonction `eval_var_expr` possédant un paramètre `e` de type `expr`, retourne la liste des variables contenues dans `e`. Cette fonction sera utilisée par le solveur, comme nous allons le voir dans la suite.

Le solveur fonctionne de cette manière : pour une contrainte donnée c et la liste des variables contenues dans c, le solveur va itérer la liste des variables pour les restreindre une à une grâce aux fonctions de restrictions du module Intervals. Plus précisément, pour restreindre les variables une par une s'il y a une quelconque opération arithmétique dans la contrainte, la variable courante est isolée dans la contrainte pour pouvoir utiliser les fonctions de restrictions.

Par exemple, supposons la contrainte $x_1 + x_2 < x_3 * x_4$, avec $x_1 = [3, 15], x_2 = [3, 13], x_3 = [2, 3]$ et $x_4 = [2, 6]$. La liste des variables est alors la suivante : $(x_1, [3, 15]) :: (x_2, [3, 13]) :: (x_3, [2, 3]) :: (x_4, [2, 6])$. En suivant l'ordre de la liste, la fonction va commencer par restreindre la variable x_1. Pour cela, il faut d'abord isoler cette variable dans la contrainte. Alors la contrainte $x_1 + x_2 < x_3 * x_4$ est transformée en $x_1 < x_3 * x_4 - x_2$. Ensuite, les opérations entre tous les autres intervalles contenus dans les variables (x_2 à x_4) sont évaluées grâce à la fonction $eval_expr$. Ici, l'évaluation de $x_3 * x_4 - x_2$ est :
$eval_expr(Sous(Mult(Variable(x_3, [2,3]), Variable(x_4, [2,6])), Variable(x_2, [3,13])))$
$= [2,3] * [2,6] - [3,13] = [-9, 15]$.

La fonction d'évaluation est ensuite appelée avec la valeur de la variable x_1 et l'évaluation des opérations entre les autres variables, *i.e.* dans ce cas on a :

$$restrict_var_varC_leqStr\ [3,15]\ [-9,15] = [3,14].$$

Puis le même algorithme est appliqué pour x_2, x_3 et x_4. Détaillons les opérations utilisées pour restreindre la variable x_4 : x_4 est isolée donc la contrainte est transformée en $(x_1 + x_2)/x_3 < x_4$. Comme x_4 est du côté droit de la contrainte, nous devons utiliser une restriction $varC_var$ (voir les explications sur la syntaxe dans la partie précédente), *i.e.* $restrict_varC_var_leqStr$. On a alors :
$eval_expr(Div(Plus(Variable(x_1, [3,15]), Variable(x_2, [3,13])), Variable(x_3, [2,3])))$
$= ([3,15] + [3,13])/[2,3] = [3,14]$.

Puis on obtient ensuite le résultat suivant : $restrict_varC_var_leqStr\ [3,14]\ [2,6] = [4,6]$. Une fois que toutes les variables sont restreintes, le solveur retourne alors la sur-approximation de la solution calculée.

Module `LsConstraints`

Le module `Constraint` est utilisé pour trouver une solution sur-approchée pour une seule contrainte. Le module `LsConstraint` retourne une solution sur-approchée pour une conjonction de plusieurs contraintes. Cette conjonction de plusieurs contraintes est modélisée par une liste de contraintes. Une sur-approximation est calculées pour chaque contraintes de la liste, et pour chaque variable, l'intersection des solutions de chaque contraintes calculées pour cette variable est retournée.

Exécution du solveur sur un exemple

Soient le système de réécriture $\mathcal{R}$ et le LTA $\mathcal{A}_0$ suivants :
$\mathcal{R} = \{f(x_1, g(x_2, x_3), h(x_4)) \to f(g(x_1, x_2), h(x_3), x_4) \Leftarrow (x_1 + x_2 < x_3 * x_4) \wedge (x_4 < 6)$,
$\mathcal{A}_0 = \{[3, 15] \to q_1,\ [3, 13] \to q_2,\ [2, 3] \to q_3,\ [2, 6] \to q_4,\ g(q_2, q_3) \to q_5,\ h(q_4) \to q_6,\ f(q_1, q_5, q_6) \to q_f\}$. Alors l'algorithme de flitrage retourne la substitution suivante : $\sigma = \{x_1 \mapsto [3, 15], x_2 \mapsto [3, 13], x_3 \mapsto [2, 3], x_4 \mapsto [2, 6]\}$.

Nous devons alors restreindre cette substitution vis à vis de la contrainte : Nous voulons une restriction qui soit une sur-approximation de la solution de la contrainte $(x_1 + x_2 < x_3 * x_4) \wedge (x_1 < 7)$. Le solveur est alors utilisé.

Tout d'abord, nous avons ici une conjonction de deux contraintes, modélisées par une liste de deux contraintes pour le module `LsConstraints`. La résolution va tout d'abord commencer par la première contrainte $(x_1 + x_2 < x_3 * x_4)$. Nous pouvons voir dans la partie **Module** `Constraint` une explication détaillée de cette résolution. On a alors la contrainte $x_1 + x_2 < x_3 * x_4$, avec $x_1 = [3, 15], x_2 = [3, 13], x_3 = [2, 3]$ et $x_4 = [2, 6]$, la liste des variables est donc la suivante : $(x_1, [3, 15]) :: (x_2, [3, 13]) :: (x_3, [2, 3]) :: (x_4, [2, 6])$, et elle est donnée en paramètre de la fonction $solve_simple$. La description de cette fonction est décrite dans la partie **Module** `Constraint`, et comme nous l'avons vu, la solution pour la variable x_1 est $[3, 14]$, et la solution pour la variable x_4 est $[4, 6]$. Pour x_2 nous procédons de la même manière. Tout d'abord transformer la contrainte en $x_2 < x_3 * x_4 - x_1$. Puis calculer l'évaluation de $x_3 * x_4 - x_1 = [2, 3] * [2, 6] - [3, 15] = [-11, 15]$. La fonction de restriction retourne alors l'intervalle $[3, 13]$ (pas de restriction nécessaire). Pour x_3 on a $(x_1+x_2)/x_4 < x_3$. Puis on calcule l'évaluation de $(x_1+x_2)/x_4 = [3, 15] + [3, 13]/[2, 6] = [1, 14]$. La fonction de restriction retourne l'intervalle $[2, 3]$ ((pas de restriction nécessaire). La solution retournée est alors la suivante : $(x_1, [3, 14]) :: (x_2, [3, 13]) :: (x_3, [2, 3]) :: (x_4, [4, 6])$.

La deuxième contrainte $x_4 < 6$ restreint la variable x_4 et la solution retournée est $(x_4, [2, 5])$. La solution finale retournée par le module `LsContraints` est une intersection des solutions trouvées pour chaque contraintes, et va donc retourner $(x_1, [3, 14]) :: (x_2, [3, 13]) :: (x_3, [2, 3]) :: (x_4, [4, 5])$. Les nouvelles transitions correspondantes vont donc être ajoutées à l'automate courant, et on a alors :
$\mathcal{A}_1 = \mathcal{A}_0 \cup \{[3, 14] \to \mathbf{q}_{S1},\ [4, 5] \to \mathbf{q}_{S4},\ g(\mathbf{q}_{S1}, q_2) \to q_7,\ h(q_3) \to q_8,\ f(q_7, q_8, \mathbf{q}_{S4}) \to q'_f,\ q'_f \to q_f\}$.

6.4 Expérimentations

Dans cette section, nous allons comparer l'efficacité de la complétion LTA par rapport à la complétion standard (décrite en section 2.4). Notons que dans le cas des programmes $Java$, les mêmes propriétés peuvent être prouvées aussi bien par Timbuk que TimbukLTA.

6.4.1 Un exemple introductif

Pour illustrer l'utilité de TimbukLTA, nous allons commencer par un exemple simple non relié à la programmation $Java$, mais dont la spécification est suffisamment petite pour être lisible. Soit $filter$ une fonction permettant d'enlever tous les 0 de n'importe quelle liste d'entiers. Voici la spécification TimbukLTA correspondante.

```
Ops filter:1 nil:0 cons:2
Vars F X Y Z U Xs

TRS R1
  filter(nil) -> nil
  filter(cons(X,Y)) -> cons(X,filter(Y)) if >(X,0)
  filter(cons(X,Y)) -> cons(X,filter(Y)) if <(X,0)
  filter(cons(X,Y)) -> filter(Y)         if =(X,0)

Automaton A0
  States qf qln qn
  Final States qf
  Transitions
      filter(qln)->qf
      nil->qln
      cons(qn,qln)->qln
      [-oo;+oo]->qn

Equations Approx
  Rules cons(X,Y)=Y
```

L'automate de départ `A0` représente une liste infinie de l'intervalle `[-oo;+oo]`. L'automate résultant de la complétion de `A0` par `R1` est le suivant :

```
States q0 q1 q2 q3 q4 q6 q7 q8
  Final States q0
  Transitions
      [-oo,+oo]->q6
      filter(q2)->q4
      cons(q7,q0)->q4
      cons(q8,q0)->q4
      nil->q2
      [-oo,+oo]->q3
      filter(q2)->q0
      cons(q6,q2)->q2
      nil->q1
      [-oo,-1]->q8
      [1,+oo]->q7
      cons(q8,q0)->q0
      cons(q7,q0)->q0
      nil->q0
```

Cet automate reconnait tous les calculs intermédiaires de la fonction $filter$ ainsi que son résultat : des listes d'entiers inclus dans l'intervalle $[-\infty, 1]$ ou dans l'intervalle $[1, +\infty]$, ce qui est le résultat attendu.

Ici, on peut appuyer ce résultat en prouvant la non-existence de certains mauvais *pattern* dans l'automate (voir section 2.5). Par exemple, imaginons que l'on veuille vérifier qu'il n'y a pas de 0 en première, deuxième ou troisième position de la liste. Il faut alors ajouter les lignes suivantes à la spécification :

```
Patterns
    cons(0,_)
    cons(_, cons(0,_))
    cons(_, cons(_, cons(0,_)))
```

Après exécution du fichier Timbuk avec le paramètre `-exact`, le résultat retourné est `Proof done!`, ce qui signifie qu'il n'a trouvé aucun des pattern interdits.

Si l'on veut vérifier qu'il n'y a aucun 0 dans les listes produites, il faut ajouter les règles de réécriture suivantes :

```
check(nil) -> true
check(cons(X,Y)) -> check(Y) if X!=0
check(cons(X,Y)) -> false if X==0
```

Ainsi, le terme `false` est produit dès qu'un 0 est trouvé à n'importe quel endroit de la liste. Pour vérifier l'absence de 0, il suffit alors d'ajouter le *pattern* interdit suivant :

```
Patterns
    false
```

En effet, si un `false` est atteint, cela signifie qu'il y a au moins un 0 dans la liste.

6.4.2 Un exemple sans arithmétique : `Threads`

Cet exemple, dénué d'opérations arithmétiques, gère plusieurs *threads* et leurs synchronisation. Voici un programme $Java$ modélisant la synchronisation de plusieurs *threads* permettant d'afficher un nombre à l'écran.

```
class T1 extends java.lang.Thread{
    private int l;
    public T1(int l){this.l=l;}
    public void run(){
        while (true){
            synchronize(Top.lock){
                System.out.println(Top.f);
                Top.f=l;
                System.out.println(Top.f);
                Top.f=0;
            }
        }
    }
}

class Top{
    public static Object lock;
    public static int f;
    public static void main(String[] argv){
        int i=1;
        lock = new Object();
        Top.f=0;
        while (i<=2){
```

```
            T1 t1 = new T1(i++);
            t1.start();
        }
    }
}
```

Depuis le *bytecode* de ce programme $Java$, Copster produit un système de réécriture de 863 règles. L'objectif de cette analyse est de prouver, quelque soit l'ordre des *threads* dans la séquence d'entiers affichés en sortie, qu'il ne peut pas y avoir deux 0 consécutifs. En d'autres mots, notre but est de prouver que la portion critique du code qui affecte la variable partagée `Top.f` est *vraiment* protégée par l'instruction $Java$ `synchronize`. Initialement, la variable `Top.f` vaut 0. Chaque *thread* affiche `Top.f` à l'écran, modifie la variable `Top.f` pour lui donner la valeur de l'entier identifiant le *thread*, affiche cette valeur à l'écran et modifie encore une fois la variable `Top.f` pour lui redonner la valeur 0. Ce procédé est répété indéfiniment. Si la synchronisation échoue, alors plusieurs *threads* peuvent exécuter cette portion critique du code en même temps et le programme est susceptible d'afficher plusieurs 0 consécutifs en sortie.

Comme les *threads* bouclent à l'infini, une équation d'approximation est nécessaire. Sur cet exemple simple, le seul terme qui puisse être amené à grandir indéfiniment est le terme représentant le flux de sortie.

Ces termes sont de la forme $outstack(x, outstack(y, \ldots))$ et représentent un flux de sortie dont le dernier élément affiché est x, avec y affiché immédiatement avant. Une simple équation d'approximation de la forme $outstack(x, outstack(y, z)) = z$ est donc suffisante pour la complétion standard, et cette complétion termine en 56 secondes après 306 étapes. Pour l'automate $\mathcal{A}^{306}_{\mathcal{R},E}$ nous pouvons facilement vérifier que le *subpattern* $outstack(zero, outstack(zero, _))$ n'est pas présent dans les termes accessibles.Autrement dit, il s'agit de vérifier qu'aucun *sous-terme* (*subpattern*) de cette forme ne peut être trouvé, *i.e.* le flux de sortie ne peut contenir à aucun moment deux $zero$ successifs.

Les éléments suivants sont donc ajoutés à la spécification Timbuk générée automatiquement depuis Copster :

```
SubPatterns
   outstack(zero,outstack(zero,_))
 /*outstack(0,outstack(0, _)) dans le cas des LTA*/

Equations Approx
  Rules outstack(x,outstack(y,z))=z
```

Pour le même exemple, le système de réécriture *conditionnel* possède 788 règles de réécriture et la complétion LTA avec TimbukLTA termine en 280 secondes après 328 étapes de complétion LTA. Nous pouvons vérifier de la même manière pour l'automate $\mathcal{A}^{328}_{\mathcal{R},E}$ qu'aucun *pattern* de la forme $outstack(0, outstack(0, _))$ n'est trouvé, prouvant que la synchronisation est bien réalisée. La complétion LTA est donc capable de prouver la même propriété. Cependant, nous pouvons également remarquer que, sur cet exemple où l'arithmétique n'est pas cruciale pour l'analyse, l'utilisation des treillis peut réduire son efficacité. Ce n'est plus du tout le cas quand l'arithmétique a son importance, comme nous allons le voir dans le prochain exemple.

6.4.3 Un exemple avec arithmétique et méthode récursive : Euclide

Cet exemple utilise uniquement des opérations arithmétiques et des appels récursifs de méthodes, et permet de montrer que la complétion LTA est plus efficace que la complétion standard dès que quelques sous-termes comportent des nombres et des opérations arithmétiques. Depuis le bytecode de ce programme $Java$, CopsterLTA produit un système de réécriture conditionnel de 695 règles.

```
public class ExEuclide {
    static int my_div( int a, int b) {//returns a/b
if (a<b)
    return 0;
else
    return 1+my_div(a-b,b);
}
    static int my_mod (int a, int b){
if (a<b)
    return a;
else
    return my_mod(a-b,b);
}
public static void main (String[] args){
    int i,j,k;
    for (i=1; i<=5; i++){
     j=my_div(10*i,7);
     k=my_mod(10*i,7);
     System.out.println(k);
    }
  }
}
```

Ce programme modélise la division euclidienne entre deux entiers a et b et retourne le résultat de cette division ainsi que son reste (*i.e.* le modulo), et affiche le reste en sortie. Sur cet exemple, aucune approximation n'est nécessaire et nous pouvons prouver que le flux de sortie contient uniquement des valeurs dans l'interval $[1; 6]$, puisque cet intervalle contient les restes de toutes les divisions des entiers 10 à 50 par l'entier 7.

La partie suivante est donc ajoutée au fichier de spécification Timbuk généré automatiquement par Copster. Dans le cas de la complétion standard :

```
SubPatterns
        outstack(zero,_)
        outstack(pred(_),_)
        outstack(succ(succ(succ(succ(succ(succ(succ(_))))))),_)
```

Dans le cas de la complétion LTA :

```
SubPatterns
        outstack([-oo;0],_)
        outstack([7;+oo],_)
```

Nous pouvons prouver le même résultat avec la complétion standard et avec la complétion LTA. Cependant, la complétion LTA nécessite seulement 14 secondes et 727 étapes alors que la complétion standard nécessite 59 secondes et 2019 étapes. Notons également que les termes interdits (les éléments `SubPatterns` décrits ci-dessus) sont modélisable de manière plus concise dans le cas des LTA.

6.4.4 Un exemple avec arithmétique et allocation d'objet : `FactoList`

Ce dernier exemple comporte de la création d'objet ainsi qu'un peu d'arithmétique. Cet exemple confirme que la complétion LTA est plus efficace que la complétion standard quand l'arithmétique est présente. Depuis le bytecode de ce programme $Java$, CopsterLTA produit un système de réécriture conditionnel de 793 règles.

```
class List{
   List next;
   int val;
   public List(int elt, List l){
      next =l;
      val= elt;
   }

   public void printList(){
      List l=this;
      while (l!=null){
            System.out.println(l.val);
            l=l.next;
      }
   }
}

public class FactoList {
   public static int factorial(int i){
      int res=1;
      for (int j=2; j<=i; j++){
           res=res*j;
      }
      return res;
   }

   public static void main(String arg[]){
      List ls= null;
      int x=0;
      while (x>=0) {
           try {x=System.in.read();} catch(java.io.IOException e){};
           if (x>=0) ls=new List(factorial(x),ls);
      }
      ls.printList();
   }
}
```

Dans ce programme, les entiers sont lus sur le canal d'entrée et leurs factorielles sont stockées dans une liste simplement chainée. À la fin, le contenu de la liste est affiché en sortie en utilisant la méthode `printList`. La complétion peut être utilisée pour montrer, par exemple, qu'aucun entier inférieur à 1 n'est affiché dans le canal de sortie, quelque soit l'entier lu sur le canal d'entrée.

Cette fois-ci nous avons besoin d'équations pour sur-approcher les entiers, le tas (qui peut contenir un nombre infini d'objets), et le canal de sortie qui peut contenir un nombre infini d'entiers. Cette approximation est effectuée en utilisant trois équations simples :

- l'équation $outstack(x, y) = y$ fusionne des termes infini de la pile de sortie dans une même classe d'équivalence,
- l'équation $stack1(_, _, _) = stack1(_, _, _)$ fusionne tous les objets de la classe `List` (qui sont stockés pdans des termes dont la racine est le symbole $stack1$ dans le système de réécriture généré),
- les équations $succ(succ(x)) = succ(x)$ et $pred(pred(x)) = pred(x)$ fusionnent tous les entiers dans trois classes d'équivalences : soit 0, soit 1 ou plus, soit -1 ou moins.

En utilisant ces équations, la complétion standard permet de produire, en 20 secondes et 467 étapes de complétion, un automate $\mathcal{A}^{467}_{\mathcal{R},E}$ ne contenant aucun terme de la forme $outstack(zero, _)$ ou $outstack(pred(_), _)$, signifiant qu'aucun entier inférieur à 1 n'est dans le flux de sortie.

Pour la complétion LTA, les équations nécessaires sont similaires pour calculer une approximation de la pile et du canal de sortie. Cependant, les équations sont légèrement différentes pour l'approximation des entiers. En effet, les valeurs d'entiers infiniment croissantes sont automatiquement capturées grâce à l'étape de *widening* décrite en sous-section 5.4.3 du chapitre précédent. Cependant, les termes construits sur les symboles *built-in* peuvent être infinis. En utilisant le précédent programme $Java$, nous constatons que les termes *built-in* infinis qui sont susceptibles d'êtres construits sont de la forme $+(+(+(_, 1), 1), 1)$ à cause de l'instruction `j++` et de la forme $*(_, *(_, *(_, _)))$ à cause de l'opération factorielle. Une approximation de ces deux types de termes peut être calculée grâce aux deux équations suivantes :

- $+(x, 1) = x$ et
- $*(x, y) = y$.

En 40 secondes et 349 étapes de complétion, un point-fixe est obtenu par complétion LTA. La même propriété, qui est, rappelons-le, qu'aucun entier inférieur à 1 n'est affiché en sortie, peut être prouvée pour l'automate $\mathcal{A}^{349}_{\mathcal{R},E}$. Étant donné que l'approximation sur les entiers est assez simple (seulement trois classes d'équivalence), la complétion standard avec entiers de Peano est meilleure en temps que la complétion LTA. Cependant, dès qu'une approximation plus fine est nécessaire sur les entiers (exigeant plus de classes d'équivalences sur les entiers), la complétion LTA obtient de meilleures performances que la complétion standard.

Si nous restreignons les valeurs des entiers sur le flux d'entrée, alors nous pouvons prouver une propriété plus précise sur les valeurs des entiers du flux de sortie. Si nous restreignons les entiers d'entrée aux valeurs contenues dans l'intervalle $[2; +\infty]$ (ou -1 pour l'arrêt de la boucle), alors nous pouvons prouver que les entiers du flux de sortie sont tous supérieurs à 1. Pour cela, nous avons également besoin de raffiner les équations d'approximation sur les entiers et créer quatre classes d'équivalences d'en-

tiers : soit -1 ou moins, soit 0, soit 1, soit 2 ou plus. Cette approximation est faite manuellement mais pourrait être automatisée en utilisant une complétion CEGAR comme dans [Boichut et al., 2012]. Ce type d'abstraction est rendu possible grâce aux équations *conditionnelles* où la condition permet d'affiner l'approximation induite par l'équation (exemple :$+(x, 1) = x \; if \; x > 2$). Ainsi, la propriété peut être prouvée en utilisant la complétion standard en 21 secondes et 468 étapes de complétion, tandis que la complétion LTA s'effectue en 14 secondes et 430 étapes de complétions.

Si nous restreignons l'intervalle des valeurs d'entrée à $[3; +\infty]$ pour prouver que les valeurs du flux de sorties sont supérieures à 5, alors la complétion standard s'effectuera en 320 secondes et 953 étapes de complétion, tandis que la complétion LTA est plus stable et s'effectuera en 15 secondes et 467 étapes de complétion. Pour un ensemble de valeurs restreint à partir de $[4; +\infty]$, la complétion standard surcharge la mémoire et doit être arrêtée après 2 heures et 1500 étapes de complétion. Alors que la complétion LTA s'effectue en 32 secondes et 641 étapes de complétion, prouvant qu'aucun entier inférieur à 24 n'est affiché sur le flux de sortie.

6.5 Conclusion et travaux futurs

Exemples	Complétion standard			Complétion LTA		
	Nb de règles	Étapes de complétion	Temps de complétion	Nb de règles	Étapes de complétion	Temps de complétion
Threads	863	306	56s	788	328	280s
Euclid	776	2019	59s	695	727	14s
FactoList input=$(3, 1, 2, 0)$	892	799	17s	793	538	33s
FactoList input=$(7, 5, 6, 4, 1)$	892	>9465	>2h	793	1251	250s
FactoList input=$[-\infty; +\infty]$	892	467	20s	793	349	40s
FactoList input=$[3; +\infty]$	892	953	320s	793	467	15s
FactoList input=$[4; +\infty]$	892	>1500	>2h	793	641	32s

TABLEAU 6.1 — Comparaisons de performances entre la complétion standard et la complétion LTA.

Nous avons montré dans ce chapitre que l'encodage incluant les treillis dans les automates d'arbres pouvait améliorer considérablement la vérification des systèmes de réécriture s'appuyant sur l'arithmétique. Ceci est illustré sur la vérification des programmes $Java$, traduits de manière automatique en systèmes de réécriture conditionnels grâce à CopsterLTA, une adaptation pour nos travaux de l'outil existant Copster. Ces programmes $Java$ sont vérifiés en utilisant TimbukLTA, une nouvelle implémentation de Timbuk intégrant la complétion LTA. L'implémentation présentée dans ce chapitre est une première instance de TimbukLTA où nous avons branché le domaine abstrait des intervalles d'entiers. Ce simple domaine abstrait a permis d'améliorer la vérification

des programmes $Java$ comportant des opérations arithmétiques sur les entiers. Le LTA résultant de la complétion combine de manière homogène les domaines abstraits pour abstraire les valeurs numériques, ainsi que que les automates d'arbres pour abstraire les structures telles que les threads, les piles, les tas et les objets.

Les expérimentations effectuées afin de prouver l'efficacité de la complétion LTA sont résumées dans le tableau 6.1, qui permet de montrer que l'intégration des LTA dans la complétion peut réduire sont efficacité quand le système de réécriture n'utilise pas d'arithmétique. À l'opposé, contrairement à la complétion standard, le temps d'exécution de la complétion LTA augmente proportionnellement et de façon stable quand l'arithmétique est nécessaire à l'analyse. Ceci est illustré dans l'exemple "Facto-List". L'analyse complète du système de réécriture modélisant la sémantique de ce programme gère un nombre très peu élevé d'opérations arithmétiques. Cependant, même pour ce nombre d'opérations arithmétiques limité, quand la précision sur la valeur numérique est nécessaire, l'utilisation des treillis dans la complétion devient nécessaire, ou quand des valeurs un peu élevées de factorielles doivent être calculées, comme nous le voyons sur le tableau 6.1 entre la liste $(3, 1, 2, 0)$ et la liste $(7, 5, 6, 4, 1)$ passées en entrée. De plus, cette implémentation permet le traitement d'équations *conditionnelles*, ce qui n'était pas le cas dans Timbuk.

Étant donné que la complétion LTA n'est pas dédiée au domaine abstrait spécifique des intervalles d'entiers, nous aimerions brancher de nombreux autres domaines abstraits dans TimbukLTA, tels que les domaines abstraits pour les chaînes de caractères et les réels. Il est aussi prévu de définir des contraintes syntaxiques sur les équations, afin de garantir la terminaison de la complétion LTA, comme définit dans [Genet, 2014].

Conclusion et perspectives

Dans ce travail de thèse, nous nous sommes intéressés à la vérification de propriétés de sûreté sur des systèmes à états possiblement infinis. La modélisation choisie représente les ensembles potentiellement infinis d'états du système par des automates d'arbres, et le comportement du système par un système de réécriture.

Pour vérifier une propriété de sûreté, il est nécessaire de calculer l'ensemble des états accessibles, afin de vérifier qu'aucun d'entre eux n'a de comportement indésirable. Pour calculer cet ensemble d'états accessibles dans le cadre de la modélisation choisie dans cette thèse, on utilise alors l'algorithme de *complétion d'automates d'arbres*. Cette méthode est *générique* et permet de modéliser de manière relativement simple de nombreux systèmes, comme nous l'avons vu dans le chapitre 3. Toutefois, la complétion d'automates d'arbres possède certaines limites que nous avons tenté d'améliorer dans cette thèse.

La première limite concerne le choix manuel de l'abstraction. Dans le chapitre 4, nous avons proposé une caractérisation automatique par des formules logiques de ce qu'était une "bonne" sur-approximation : une sur-approximation n'admettant pas de faux contre-exemple, sans que l'utilisateur n'ait à fournir manuellement une fonction d'abstraction. Cette caractérisation est rendue possible grâce à un nouveau type d'automate d'arbres contenant des variables, appelé STA (*Symbolic Tree Automaton*). Nous avons également défini des algorithmes de recherche de cette bonne sur-approximation, et prouvé, pour l'un d'entre eux, que si le système admettait un point-fixe régulier n'admettant aucun terme interdit, alors il serait nécessairement trouvé par l'algorithme. Grâce à cela, cette méthode constitue une procédure de décision permettant de répondre à la question : "existe-t-il une sur-approximation vérifiant la propriété possédant au plus n états ?" Cependant, la limite principale de cette méthode réside dans la taille très importante des formules de caractérisation générées rendant pour l'instant difficile la vérification de problèmes concrets. La grande taille des formules générées donne toutefois une information importante : elle permet de donner une indication concrète de la complexité du problème d'accessibilité dans le cas de systèmes à états infinis, et de quantifier la difficulté du problème auquel on s'attaque dans cette thèse.

La seconde limite concerne le temps de calcul parfois élevé du calcul de complétion quand on s'attaque à des problèmes de la vie courante. La difficulté du passage à l'échelle peut être notamment due au traitement des domaines infinis représentés, tels que l'ensemble $\mathbb{Z}$: les premiers travaux sur la vérification de programmes $Java$ à l'aide de la complétion utilisent en effet une représentation en entier de Peano ralentissant considérablement le calcul des opérations arithmétiques. Dans le chapitre 5, nous avons alors fourni la possibilité d'intégrer un domaine infini dans l'algorithme de

complétion, et simplifié de beaucoup le calcul des opérations associées à ce domaine grâce à une étape d'évaluation. Afin d'intégrer un domaine infini dans l'algorithme de complétion et pouvoir traiter le comportement infini résultant de cette intégration, nous avons alors défini un nouveau type d'automate d'arbres enrichi d'éléments d'un treillis abstrait et de ses opérations : les LTA (*Lattice Tree Automata*). Nous avons alors conçu l'algorithme de complétion associé, utilisant :

- un solveur,
- une étape d'évaluation,
- un nouveau type d'équations d'abstraction, pouvant être *conditionnelles*, et
- un opérateur de *widening*.

Le principal intérêt de ce travail est sa généricité : en effet cette technique est paramétrable et offre la possibilité de brancher n'importe quel treillis atomique et unidimensionnel dans l'algorithme de complétion.

L'efficacité de cette méthode est démontrée dans le chapitre 6, où différentes expérimentations y sont décrites. Ces expérimentations utilisent de nouvelles implémentations de Timbuk et Copster adaptées à la complétion des LTA : TimbukLTA et CopsterLTA, disponibles sur les pages http://www.irisa.fr/celtique/genet/timbuk/ et http://www.irisa.fr/celtique/genet/copster/. Les expérimentations effectuées permettent de montrer que le temps d'exécution de la complétion LTA augmente proportionnellement et de façon stable quand l'arithmétique est nécessaire à l'analyse, tandis que l'augmentation des valeurs numériques peut faire exploser le calcul dans le cas de la complétion standard (*i.e.* sans LTA).

Perspectives

De nombreuses perspectives apparaissent à la suite de ce travail de thèse. Le problème de la méthode décrite dans le chapitre 4 est la taille trop importante des formules générées. Dans ce cas deux possibilités se présentent : enrichir la méthode de techniques de résolutions de contraintes dédiées et d'heuristiques qui nous permettraient de manipuler des formules de taille importante, ou réduire la taille des formules lors de leur calcul en éliminant des branches inutiles à la volée. La formule la plus importante en terme de taille est la formule point-fixe $\phi^{FP}_{\mathcal{R},\mathcal{A}_S}$. Une idée serait alors, en se basant sur la formule $\phi^{Bad}_{\mathcal{A}_S}$ générée au préalable, de construire la formule $\phi^{FP}_{\mathcal{R},\mathcal{A}_S}$ d'une manière plus intelligente. Cette formule se construit principalement grâce à l'algorithme de filtrage : il s'agirait, lors de la construction des formules durant l'algorithme, couper des branches inconsistantes à la volée en propageant les contraintes des termes interdits. Pour cela, une technique *à la Mona* [Henriksen et al., 1995] est en cours d'étude.

Une autre perspective à ce travail est le traitement d'un ensemble *infini* de termes interdits, que notre méthode ne permet pas à l'heure actuelle. Dans ce cas, la formule $\phi^{Bad}_{\mathcal{A}_S}$ n'est plus générée par la condition de reconnaissance d'un terme interdit dans le STA, mais plutôt par l'intersection entre le STA et l'automate $\mathcal{A}_{Bad}$ permettant de modéliser un ensemble possiblement infini de termes interdits. Le travail effectué dans [Boichut et al., 2012] pourrait alors être utilisé : en effet, cet article mentionne une description de l'intersection de deux automates par une formule logique.

Les travaux du chapitre 5 ne permettent pour le moment que d'intégrer des treillis *unidimensionnels* à l'algorithme de complétion. En tant que perspective à ces travaux,

l'intégration de treillis *relationnels* (tels que des polyèdres) est en cours de réalisation. L'intégration de tels treillis implique que chaque élément du treillis doit être défini de façon globale, autrement dit pour l'ensemble de l'automate, et non pas sur un seul état comme c'est le cas dans le chapitre 5. Le treillis est alors défini sur les états de l'automate qui deviennent des variables (de la même manière que les STA du chapitre 4). Imaginons par exemple l'ensemble de transitions Δ_0, enrichi du polyèdre P_0 suivant :

$$\Delta_0 = \{f(X_0, X_1) \to X_3\} \text{ et } P_0 = \{X_0 + X_1 < 4\}.$$

Supposons que l'on ait le système de réécriture $\mathcal{R} = \{f(x, y) \to g(x) \;\Leftarrow\; x < y\}$. Alors les transitions ajoutées doivent respecter les conditions de la règle de réécriture, mais ne doivent pas restreindre les termes reconnus par l'automate précédent. La nouvelle transition ajoutée doit contenir de nouveaux états sur lesquels seront définies les nouvelles contraintes. Ici, la transition à ajouter devrait être $g(X_0) \to X_2$, avec la contrainte $X_0 < X_1$. Mais si cette contrainte s'ajoute à P_0 sans être modifiée (*i.e.*, définie sur des états existants déjà dans l'automate), les transitions initiales de $\mathcal{A}_0$ seront davantage contraintes que précédemment. L'idée est donc d'ajouter la transition avec une variable copie X'_0 et de faire une copie du polyèdre courant avec des variables copies afin de garder les anciennes contraintes sur les nouvelles transitions. L'automate $\mathcal{A}_1$ successeur obtenu possède alors l'ensemble de transitions suivant :

$$\Delta_1 = \left\{ \begin{array}{l} f(X_0, X_1) \to X_2, \\ g(X'_0) \to q_f \end{array} \right\} \qquad P_1 = \left\{ \begin{array}{ll} & X_0 + X_1 < 4 \\ \wedge & X'_0 + X'_1 < 4 \\ \wedge & X'_0 < X'_1 \end{array} \right\}$$

On peut constater ici que la variable X'_1 ne correspond à rien dans l'automate. Il est donc intéressant de faire une projection pour garder uniquement les états présents dans l'automate :

$$\exists X'_1 \; t.q. \; X'_0 + X'_1 < 4, \exists X'_1 \; t.q. \; X'_0 < X'_1$$

$$\frac{\begin{array}{c} X'_0 + X'_1 < 4 \\ X'_0 - X'_1 < 0 \end{array}}{X'_0 < 4}$$

Après projection, on obtient alors le polyèdre suivant :

$$P_0 = \{X_0 + X_1 < 4 \wedge X'_0 < 4\}$$

L'ajout de termes *built-in* se fait dans le polyèdre et non pas dans l'automate. Ceci permet de séparer la partie "fonctionnelle" (termes classiques) de la partie "interprétation abstraite" (éléments du domaine infini et leurs opérations). Ainsi la règle $g(x) \to g(x+1)$ rajoute la règle $g(X_3) \to X'_2$ dans l'automate et la condition $(X_3 = X''_1 + 1)$ dans le polyèdre. Il reste ensuite à traiter le comportement infini du polyèdre.

On constate également que les automates enrichis de polyèdres constituent des automates à variables de la même manière que les STA du chapitre 4. De plus, les STA permettent un traitement de formules logiques sur ces états. On pourrait alors imaginer mélanger ces deux contributions : les contraintes définissables sur les STA seraient alors enrichies des contraintes correspondant au polyèdre.

Le chapitre 6 décrit l'implémentation actuelle de l'algorithme de complétion LTA, mais ne permet pour l'instant que d'intégrer le domaine des entiers relatifs et de l'abstraire par l'ensemble des intervalles sur $\mathbb{Z}$. Une perspective à ce travail est d'enrichir cette implémentation afin de brancher dans TimbukLTA de nombreux autres treillis, permettant par exemple d'abstraire les chaînes de caractères et les réels.

Des travaux récents permettent de prouver la *terminaison* de la complétion pour certaines classes d'équations, respectant une certaine condition [Genet, 2014]. Une autre perspective serait alors d'ajouter, en plus des contraintes déjà possibles sur les équations, des contraintes syntaxiques permettant de garantir la terminaison du calcul de complétion.

"Mais... Chanter, rêver, rire, passer, être seul, être libre,
Avoir l'œil qui regarde bien, la voix qui vibre,
Mettre, quand il vous plaît, son feutre de travers,
Pour un oui, pour un non, se battre, — ou faire un vers
Travailler sans souci de gloire ou de fortune,
A tel voyage, auquel on pense, dans la lune !
N'écrire jamais rien qui de soi ne sortît,
Et modeste, d'ailleurs, se dire : « Mon petit,
Sois satisfait des fleurs, des fruits, même des feuilles
Si c'est dans ton jardin à toi que tu les cueilles ! »
Puis, s'il advient d'un peu triompher, par hasard,
Ne pas être obligé d'en rien rendre à César,
Vis-à-vis de soi-même en garder le mérite,
Bref, dédaignant d'être le lierre parasite,
Lors même qu'on n'est pas le chêne ou le tilleul,
Ne pas monter bien haut, peut-être, mais tout seul..."

Edmond Rostand, *Cyrano de Bergerac*

Bibliographie

[Abdulla et al., 2007] Abdulla, P. A., Delzanno, G., et Rezine, A. (2007). Parameterized verification of infinite-state processes with global conditions. In *CAV*.

[Abdulla et al., 2008] Abdulla, P. A., Henda, N. B., Delzanno, G., Haziza, F., et Rezine, A. (2008). Parameterized tree systems. In *FORTE*.

[Abdulla et al., 2002] Abdulla, P. A., Jonsson, B., Mahata, P., et d'Orso, J. (2002). Regular tree model checking. In *Computer Aided Verification, 14th International Conference, CAV 2002,Copenhagen, Denmark, July 27-31, 2002, Proceedings*, volume 2404 of *Lecture Notes in Computer Science*, pages 555–568. Springer.

[Abdulla et al., 2005] Abdulla, P. A., Legay, A., d'Orso, J., et Rezine, A. (2005). Simulation-based iteration of tree transducers. In *Tools and Algorithms for the Construction and Analysis of Systems, 11th International Conference, TACAS 2005, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2005, Edinburgh, UK, April 4-8, 2005, Proceedings*, volume 3440 of *Lecture Notes in Computer Science*, pages 30–44. Springer.

[Armando et al., 2005] Armando, A., Basin, D., Boichut, Y., Chevalier, Y., Compagna, L., Cuellar, J., Hankes Drielsma, P., Héam, P.-C., Kouchnarenko, O., Mantovani, J., Mödersheim, S., von Oheimb, D., Rusinowitch, M., Santos Santiago, J., Turuani, M., Viganò, L., et Vigneron, L. (2005). The AVISPA Tool for the automated validation of internet security protocols and applications. In *CAV'2005*, volume 3576 of *LNCS*, pages 281–285. Springer. http://www.avispa-project.org.

[Asavoae et Asavoae, 2010] Asavoae, I. M. et Asavoae, M. (2010). Collecting semantics under predicate abstraction in the K framework. In Ölveczky, P. C., editor, *WRLA*, volume 6381 of *Lecture Notes in Computer Science*, pages 123–139. Springer.

[Baader et Nipkow, 1998] Baader, F. et Nipkow, T. (1998). *Term Rewriting and All That*. Cambridge University Press.

[Bach et al., 2012] Bach, J.-C., Crégut, X., Moreau, P.-E., et Pantel, M. (2012). Model transformations with tom. In *LDTA*, page 16. ACM.

[Bae et al., 2013] Bae, K., Escobar, S., et Meseguer, J. (2013). Abstract logical model checking of infinite-state systems using narrowing. In *RTA*, volume 21 of *LIPIcs*, pages 81–96. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik.

[Ball et Rajamani, 2000] Ball, T. et Rajamani, S. K. (2000). Bebop: A symbolic model checker for boolean programs. In *SPIN*, volume 1885 of *Lecture Notes in Computer Science*, pages 113–130. Springer.

[Ball et Rajamani, 2001] Ball, T. et Rajamani, S. K. (2001). The slam toolkit. In *CAV*, volume 2102 of *Lecture Notes in Computer Science*, pages 260–264. Springer.

[Balland et al., 2008] Balland, E., Boichut, Y., Genet, T., et Moreau, P.-E. (2008). Towards an Efficient Implementation of Tree Automata Completion. In *AMAST'08*, volume 5140 of *LNCS*. Springer.

[Balland et al., 2007] Balland, E., Moreau, P.-E., et Reilles, A. (2007). Bytecode rewriting in tom. *Electr. Notes Theor. Comput. Sci.*, 190(1):19–33.

[Barré et al., 2009] Barré, N., Besson, F., Genet, T., Hubert, L., et Le Roux, L. (2009). Copster homepage. http://www.irisa.fr/celtique/genet/copster.

[Bauer et al., 2011] Bauer, S., Fahrenberg, U., Juhl, L., Larsen, K., Legay, A., et Thrane, C. (2011). Quantitative refinement for weighted modal transition systems. In *MFCS*, volume 6907 of *LNCS*. springer.

[Bertot et Castéran, 2004] Bertot, Y. et Castéran, P. (2004). Interactive theorem proving and program development. coq'art: The calculus of inductive constructions. http://coq.inria.fr/.

[Blanchet et al., 2002] Blanchet, B., Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., et Rival, X. (2002). Design and implementation of a special-purpose static program analyzer for safety-critical real-time embedded software. In *The Essence of Computation*, volume 2566 of *Lecture Notes in Computer Science*, pages 85–108. Springer.

[Boichut, 2005] Boichut, Y. (2005). T44sp. http://www.univ-orleans.fr/lifo/Members/Yohan.Boichut/ta4sp.html.

[Boichut et al., 2012] Boichut, Y., Boyer, B., Genet, T., et Legay, A. (2012). Equational Abstraction Refinement for Certified Tree Regular Model Checking. In *ICFEM'12*, volume 7635 of *LNCS*. Springer.

[Boichut et al., 2008a] Boichut, Y., Courbis, R., Heam, P.-C., et Kouchnarenko, O. (2008a). Finer is better: Abstraction refinement for rewriting approximations. In *RTA*, LNCS 5117, pages 48–62. Springer.

[Boichut et al., 2009] Boichut, Y., Courbis, R., Héam, P.-C., et Kouchnarenko, O. (2009). Handling non left-linear rules when completing tree automata. *IJFCS*, 20(5).

[Boichut et al., 2006] Boichut, Y., Genet, T., Jensen, T., et Le Roux, L. (2006). Rewriting Approximations for Fast Prototyping of Static Analyzers. Research Report RR 5997, INRIA.

[Boichut et al., 2007] Boichut, Y., Genet, T., Jensen, T., et Leroux, L. (2007). Rewriting Approximations for Fast Prototyping of Static Analyzers. In *RTA*, LNCS 4533, pages 48–62.

[Boichut et Héam, 2008] Boichut, Y. et Héam, P.-C. (2008). A theorerical limit for safety verification techniques with regular fix-point computations. *Information Processing Letters*, 108(1):1–2.

[Boichut et al., 2008b] Boichut, Y., Héam, P.-C., et Kouchnarenko, O. (2008b). Approximation-based tree regular model-checking. *Nord. J. Comput.*, 14(3):216–241.

[Boigelot et al., 2003] Boigelot, B., Legay, A., et Wolper, P. (2003). Iterating transducers in the large (extended abstract). In *Computer Aided Verification, 15th International Conference, CAV 2003, Boulder, CO, USA, July 8-12, 2003, Proceedings*, volume 2725 of *Lecture Notes in Computer Science*, pages 223–235. Springer.

[Bouajjani et al., 2006a] Bouajjani, A., Habermehl, P., Rogalewicz, A., et Vojnar, T. (2006a). Abstract regular tree model checking. *Electron. Notes Theor. Comput. Sci.*, 149:37–48.

[Bouajjani et al., 2006b] Bouajjani, A., Habermehl, P., Rogalewicz, A., et Vojnar, T. (2006b). Abstract Regular Tree Model Checking of Complex Dynamic Data Structures. In *SAS'06*, volume 4134 of *LNCS*, pages 52–70. Springer.

[Bouajjani et al., 2004] Bouajjani, A., Habermehl, P., et Vojnar, T. (2004). Abstract regular model checking. In *Computer Aided Verification, 16th International Conference, CAV 2004, Boston, MA, USA, July 13-17, 2004, Proceedings*, volume 3114 of *Lecture Notes in Computer Science*, pages 372–386. Springer.

[Bouajjani et al., 2000] Bouajjani, A., Jonsson, B., Nilsson, M., et Touili, T. (2000). Regular model checking. In *Computer Aided Verification, 12th International Conference, CAV 2000, Chicago, IL, USA, July 15-19, 2000, Proceedings*, volume 1855 of *Lecture Notes in Computer Science*, pages 403–418. Springer.

[Bouajjani et Touili, 2002] Bouajjani, A. et Touili, T. (2002). Extrapolating tree transformations. In *CAV*, volume 2404 of *LNCS*. Springer.

[Bouhoula et al., 2000] Bouhoula, A., Jouannaud, J.-P., et Meseguer, J. (2000). Specification and proof in membership equational logic. *Theor. Comput. Sci.*, 236(1-2):35–132.

[Boyer et Genet, 2009] Boyer, B. et Genet, T. (2009). Verifying temporal regular properties of abstractions of term rewriting systems. In *Proceedings Tenth International Workshop on Rule-Based Programming, RULE 2009, Brasília, Brazil, 28th June 2009*, volume 21 of *EPTCS*, pages 99–108.

[Boyer et al., 2008] Boyer, B., Genet, T., et Jensen, T. P. (2008). Certifying a tree automata completion checker. In *IJCAR*, pages 523–538.

[Bruneton et al., 2002] Bruneton, E., Lenglet, R., et Coupaye, T. (2002). Asm: A code manipulation tool to implement adaptable systems. In *In Adaptable and extensible component systems*.

[Bruni et Meseguer, 2003] Bruni, R. et Meseguer, J. (2003). Generalized rewrite theories. In *Automata, Languages and Programming, 30th International Colloquium, ICALP 2003, Eindhoven, The Netherlands, June 30 - July 4, 2003. Proceedings*, volume 2719 of *Lecture Notes in Computer Science*, pages 252–266. Springer.

[Clavel et al., 2001] Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., et Quesada, J. F. (2001). Maude: Specification and programming in rewriting logic. *Theoretical Computer Science*.

[Clavel et al., 2009] Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., et Quesada, J. F. (2009). Maude homepage. http://maude.cs.uiuc.edu.

[Clavel et al., 2007] Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., et Talcott., C. L. (2007). *All About Maude, A High-Performance Logical Framework*, volume 4350 of *Lecture Notes in Computer Science*. Springer.

[Comon et al., 2007] Comon, H., Dauchet, M., Gilleron, R., Löding, C., Jacquemard, F., Lugiez, D., Tison, S., et Tommasi, M. (2007). Tree automata techniques and applications.

[Cousot et Cousot, 1977] Cousot, P. et Cousot, R. (1977). Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL*, pages 238–252.

[Cuoq et al., 2012] Cuoq, P., Kirchner, F., Kosmatov, N., Prevosto, V., Signoles, J., et Yakobowski, B. (2012). Frama-c - a software analysis perspective. In *SEFM*, volume 7504 of *Lecture Notes in Computer Science*, pages 233–247. Springer.

[David et al., 2011] David, C., Libkin, L., et Tan, T. (2011). Efficient reasoning about data trees via integer linear programming. In *ICDT*, pages 18–29.

[Durán et al., 2008] Durán, F., Lucas, S., et Meseguer, J. (2008). Mtt: The maude termination tool (system description). In *Automated Reasoning, 4th International Joint Conference, IJCAR 2008, Sydney, Australia, August 12-15, 2008, Proceedings*, volume 5195 of *Lecture Notes in Computer Science*, pages 313–319. Springer.

[Durán et Meseguer, 2010a] Durán, F. et Meseguer, J. (2010a). A church-rosser checker tool for conditional order-sorted equational maude specifications. In *Rewriting Logic and Its Applications - 8th International Workshop, WRLA 2010, Held as a Satellite Event of ETAPS 2010, Paphos, Cyprus, March 20-21, 2010, Revised Selected Papers*, volume 6381 of *Lecture Notes in Computer Science*, pages 69–85. Springer.

[Durán et Meseguer, 2010b] Durán, F. et Meseguer, J. (2010b). A maude coherence checker tool for conditional order-sorted rewrite theories. In *Rewriting Logic and Its Applications - 8th International Workshop, WRLA 2010, Held as a Satellite Event of ETAPS 2010, Paphos, Cyprus, March 20-21, 2010, Revised Selected Papers*, volume 6381 of *Lecture Notes in Computer Science*, pages 86–103. Springer.

[Engelfriet, 1975] Engelfriet, J. (1975). Bottom-up and top-down tree transformations - a comparison. *Mathematical Systems Theory*, 9(3):198–231.

[Ésik et Liu, 2007] Ésik, Z. et Liu, G. (2007). Fuzzy tree automata. *Fuzzy Sets Syst.*, 158:1450–1460.

[Farzan et al., 2004] Farzan, A., Chen, C., Meseguer, J., et Rosu, G. (2004). Formal analysis of java programs in javafan. In *CAV*, volume 3114 of *Lecture Notes in Computer Science*, pages 501–505. Springer.

[Feuillade et al., 2004] Feuillade, G., Genet, T., et Viet Triem Tong, V. (2004). Reachability Analysis over Term Rewriting Systems. *Journal of Automated Reasonning*, 33 (3-4):341–383.

[Figueira et Segoufin, 2011] Figueira, D. et Segoufin, L. (2011). Bottom-up automata on data trees and vertical xpath. In *STACS*.

[Gallagher et Rosendahl, 2008] Gallagher, J. et Rosendahl, M. (2008). Approximating term rewriting systems: a horn clause specification and its implementation. In *LPAR'08*, volume 5330. Springer.

[Genest et al., 2010] Genest, B., Muscholl, A., et Wu, Z. (2010). Verifying recursive active documents with positive data tree rewriting. In *FSTTCS*.

[Genet, 1998] Genet, T. (1998). Decidable approximations of sets of descendants and sets of normal forms. In *RTA*, volume 1379 of *LNCS*. springer.

[Genet, 2008] Genet, T. (2008). Timbuk 3.0 – a Tree Automata Completion Tool. IRISA / Université de Rennes 1. http://www.irisa.fr/celtique/genet/timbuk/.

[Genet, 2009] Genet, T. (2009). Reachability analysis of rewriting for software verification. Université de Rennes 1. Habilitation.

[Genet, 2014] Genet, T. (2014). Towards static analysis of functional programs using tree automata completion. In *WRLA'14 : 10th International Workshop on Rewriting Logic and its Applications*.

[Genet et Klay, 2000] Genet, T. et Klay, F. (2000). Rewriting for Cryptographic Protocol Verification. In *cade*, volume 1831 of *lnai*. Springer Verlag.

[Genet et Rusu, 2010] Genet, T. et Rusu, V. (2010). Equational approximations for tree automata completion. *Journal of Symbolic Computation*, 45(5):574–597.

[Genet et Salmon, 2013] Genet, T. et Salmon, Y. (2013). Tree Automata Completion for Static Analysis of Functional Programs. Technical report, INRIA. http://hal.archives-ouvertes.fr/hal-00780124/PDF/main.pdf.

[Genet et Viet Triem Tong, 2001a] Genet, T. et Viet Triem Tong, V. (2001a). Reachability Analysis of Term Rewriting Systems with Timbuk. volume 2250, pages 691–702. Springer Verlag.

[Genet et Viet Triem Tong, 2001b] Genet, T. et Viet Triem Tong, V. (2001b). Timbuk – a Tree Automata Library. IRISA / Université de Rennes 1. http://www.irisa.fr/celtique/genet/timbuk/.

[Henriksen et al., 1995] Henriksen, J., Jensen, J., Jørgensen, M., Klarlund, N., Paige, B., Rauhe, T., et Sandholm, A. (1995). Mona: Monadic second-order logic in practice. In *TACAS '95*, volume 1019 of *LNCS*.

[Henzinger et al., 2003] Henzinger, T. A., Jhala, R., Majumdar, R., et Sutre, G. (2003). Software verification with blast. In *SPIN*, volume 2648 of *Lecture Notes in Computer Science*, pages 235–239. Springer.

[Hoare, 1969] Hoare, C. A. R. (1969). An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580.

[Hubert et al., 2010] Hubert, L., Barré, N., Besson, F., Demange, D., Jensen, T. P., Monfort, V., Pichardie, D., et Turpin, T. (2010). Sawja: Static analysis workshop for java. In *Formal Verification of Object-Oriented Software - International Conference, FoVeOOS 2010, Paris, France, June 28-30, 2010, Revised Selected Papers*, volume 6528 of *Lecture Notes in Computer Science*, pages 92–106. Springer.

[Joshi et al., 1975] Joshi, A. K., Levy, L. S., et Takahashi, M. (1975). Tree adjunct grammars. *J. Comput. Syst. Sci.*, 10(1):136–163.

[Kaplan et Choppy, 1989] Kaplan, S. et Choppy, C. (1989). Abstract rewriting with concrete operations. In *RTA*, pages 178–186.

[Kupferman et Lustig, 2007] Kupferman, O. et Lustig, Y. (2007). Lattice automata. In *VMCAI*.

[Le Gall et Jeannet, 2007] Le Gall, T. et Jeannet, B. (2007). Lattice automata: A representation for languages on infinite alphabets, and some applications to verification. In *SAS'07*, volume 4634 of *LNCS*, pages 52–68. Springer.

[Leroux, 2008] Leroux, J. (2008). Structural Presburger digit vector automata. *TCS*, 409(3).

[Leroy et al., 2000] Leroy, X., Doligez, D., Garrigue, J., Rémy, D., et Vouillon, J. (2000). The Objective Caml system release 3.00 – Documentation and user's manual. http://caml.inria.fr/ocaml/htmlman/.

[Lind-Nielsen, 2002] Lind-Nielsen, J. (2002). Buddy 2.4. http://buddy.sourceforge.net.

[Meseguer et al., 2003] Meseguer, J., Palomino, M., et Martì-Oliet, N. (2003). Equational Abstractions. In *Proc. 19th CADE Conf., Miami Beach (Fl., USA)*, volume 2741 of *LNCS*, pages 2–16. Springer.

[Otto et al., 2010] Otto, C., Brockschmidt, M., von Essen, C., et Giesl, J. (2010). Automated termination analysis of java bytecode by term rewriting. In *RTA*, LIPIcs. Dagstuhl.

[Paulson, 2008] Paulson, L. C. (2008). The isabelle reference manual.

[Pichardie, 2005] Pichardie, D. (2005). *Interprétation abstraite en logique intuitionniste : extraction d'analyseurs Java certifiés*. PhD thesis, Université Rennes 1. In french.

[Regehr et al., 2012] Regehr, J., Chen, Y., Cuoq, P., Eide, E., Ellison, C., et Yang, X. (2012). Test-case reduction for C compiler bugs. In *33rd Conference on Programming Language Design and Implementation (PLDI'12)*, pages 335–346. ACM.

[Roşu, 2006] Roşu, G. (2006). K: a Rewrite-based Framework for Modular Language Design, Semantics, Analysis and Implementation. Technical Report UIUCDCS-R-2006-2802, Computer Science Department, University of Illinois at Urbana-Champaign.

[Roşu et Ştefănescu, 2011] Roşu, G. et Ştefănescu, A. (2011). Matching Logic: A New Program Verification Approach (NIER Track). In *ICSE'11: Proceedings of the 30th International Conference on Software Engineering*, pages 868–871. ACM.

[Rosu et al., 2009] Rosu, G., Schulte, W., et Serbanuta, T. F. (2009). Runtime verification of C memory safety. In Bensalem, S. et Peled, D. A., editors, *Runtime Verification (RV'09)*, volume 5779 of *Lecture Notes in Computer Science*, pages 132–152.

[Rosu et Serbanuta, 2013] Rosu, G. et Serbanuta, T. F. (2013). K overview and simple case study. In *Proceedings of International K Workshop (K'11)*, ENTCS. Elsevier. www.kframework.org/.

[SmartRight, 2001] SmartRight (2001). Smartright Technical White Paper v1.0. Thomson. http://www.smartright.org.

[Takai, 2004] Takai, T. (2004). A Verification Technique Using Term Rewriting Systems and Abstract Interpretation. In *Proc. 15th RTA Conf., Aachen (Germany)*, volume 3091 of *LNCS*, pages 119–133. Springer.

[Takai et al., 2000] Takai, T., Kaji, Y., et Seki, H. (2000). Right-linear finite-path overlapping term rewriting systems effectively preserve recognizability. In *RTA'11*, volume 1833 of *LNCS*. Springer Verlag.

[Tom, 2008] Tom (2008). Tom Homepage. http://tom.loria.fr.

[Touili, 2003] Touili, T. (2003). *Analyse symbolique de systèmes infinis basée sur les automates : application à la vérification de systèmes paramétrés et dynamiques*. PhD thesis. Informatique Paris 7 2003.

VU :

Le Directeur de Thèse
(Nom et Prénom)

VU :

Le Responsable de l'École Doctorale

VU pour autorisation de soutenance

Rennes, le

Le Président de l'Université de Rennes 1

Guy CATHELINEAU

VU après soutenance pour autorisation de publication :

Le Président de Jury,
(Nom et Prénom)