HAL
open science

Reachability analysis of rewriting for software verification

Thomas Genet

To cite this version:

Thomas Genet. Reachability analysis of rewriting for software verification. Software Engineering [cs.SE]. Université Rennes 1, 2009. tel-00477013v2

HAL Id: tel-00477013

https://theses.hal.science/tel-00477013v2

Submitted on 4 May 2010

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.

HABILITATION À DIRIGER DES RECHERCHES

présentée devant

L'Université de Rennes 1
Spécialité : informatique

par

Thomas Genet

Reachability analysis of rewriting for software verification

soutenue le 30 novembre 2009 devant le jury composé de :

M.	Olivier Ridoux	Président
MM	Ahmed Bouajjani	Rapporteurs
	John Gallagher	
	Michael Rusinowitch	
MM	Hubert Comon-Lundh	Examinateurs
	Thomas Jensen	

Remerciements

Je remercie Ahmed Bouajjani, John Gallagher et Michael Rusinowitch de s'être penchés sur ce travail et d'avoir accepté d'en être les rapporteurs. Dans la mesure où ce travail tente de combiner certaines caractéristiques du model-checking régulier et de l'analyse statique par interprétation abstraite dans un outil de preuve basé sur la réécriture, il était naturel que je vous demande d'en être les rapporteurs. Cependant, je vous remercie d'avoir accepté *ensemble* de vous acquitter de cette tâche. Vos retours respectifs me seront précieux. Je remercie, en plus, John d'avoir fait l'effort de venir depuis Roskilde même s'il a, au change, gagné quelques degrés et quelques heures de soleil!

Je remercie Hubert Comon-Lundh de participer au jury. C'est, pour moi, un peu un retour aux sources car, si je me souviens bien, j'ai présenté les premiers balbutiements de ces travaux pendant un séminaire que tu organisais. Tu m'avais d'ailleurs gentiment indiqué de précieuses références, notamment celles des travaux de Florent Jacquemard, que j'avais complètement manquées. Merci, donc, d'apporter à ce jury ton expertise dans le domaine des automates d'arbres et des systèmes de réécriture ainsi que dans le domaine de la vérification des protocoles cryptographiques qui est un des nombreux violons d'Ingres que tu partages avec Michael.

Je suis très heureux qu'Olivier Ridoux soit dans ce jury. Tout d'abord parce que, Olivier, je suis fan de ton esprit curieux et inventif (et je ne suis pas le seul). Une seconde raison est que je pense que la soutenance sera l'occasion pour toi de me poser des questions qui risquent de me tarabuster pendant quelques temps. Enfin, j'espérais pouvoir m'enorgueillir d'avoir le seul président de jury en chemise hawaïenne. Cependant, il semble que tes mandats de directeur de l'IFSIC et bientôt de l'ESIR aient eu raison de ta garde-robe, à mon grand désespoir (à nouveau, je ne suis pas le seul).

Un grand merci à Thomas Jensen d'avoir constamment (mais délicatement!) oeuvré pour que je rédige ce document et soutienne cette habilitation. Merci aussi, Thomas, de m'avoir fait profiter de ta vision de l'interprétation abstraite afin de guider mes tentatives de rapprochement avec les approximations régulières en réécriture. D'autre part, je profite de l'occasion pour te remercier de ton accueil chaleureux dans l'équipe Lande (maintenant Celtique). Pendant toutes ces années, notamment en co-encadrant des doctorants, j'ai tenté de saisir la Jensen's touch: rigueur et fermeté, le tout enrobé d'une grande dose d'humanité. J'espère pouvoir, à mon tour, utiliser et transmettre cet enseignement. Merci également d'avoir *insisté* pour participer au jury, ça fait toujours plaisir.

Je remercie tous mes collègues de l'IRISA et de l'IFSIC qui contribuent à faire de ces deux instituts des endroits où il fait bon vivre, travailler et enseigner. Il n'est pas possible de remercier nominativement tout le monde. Je pense, en particulier, à mes nombreux collègues de l'IFSIC. Merci à tous pour votre bonne humeur, cela aide à se lever le matin.

A l'IRISA, je peux difficilement ne pas remercier Lydie Mabil qui sait anticiper, quand il en est encore temps, ou résoudre, quand il est trop tard, tous nos problèmes, gros et petits. Le tout dans la bonne humeur. Chapeau. Merci aussi à tous mes collègues des équipes Celtique, Lis et Vertecs pour les discussions impromptues, les séminaires au vert et les, irremplaçables, pauses du midi.

Je remercie, particulièrement, mes anciens voisins de bureau: Michael Périn, Frédéric Besson, David Pichardie, Olivier Heen et maintenant David Cachera pour leur patience. Pas toujours facile de partager un bureau avec un batteur compulsif, désolé.

Je remercie Vlad Rusu pour sa curiosité, sa ténacité et son talent. J'espère que notre collaboration, initiée à l'occasion d'un des articles couvert par ce document, se poursuivra. Je remercie Olivier Heen de s'être intéressé à ces travaux même si ceux-ci ne sont pas toujours très "sécu". Merci aussi pour cette passionnante collaboration autour de la valorisation des outils de vérification de protocoles cryptographiques pour Thomson R&D. Je me dois, aussi, de remercier Francis Klay avec qui j'ai fait mes premières armes sur ce sujet à France Telecom avant d'être recruté à l'IRISA.

Je remercie mes trois doctorants: Valérie Viet Triem Tong, Luka Le Roux et Benoît Boyer d'avoir essuyé les plâtres. J'espère qu'ils n'auront pas trop de séquelles après démoulage. Merci aussi à Guillaume Feuillade dont j'ai eu la chance d'encadrer le stage de DEA. J'espère, simplement, qu'on arrivera un jour à finir ce fichu papier sur la complétude. Merci à tous les membres de l'ANR RAVAJ d'avoir cru et contribué à ce projet. La façon dont cela s'est passé donne presque envie de re-signer! Dans cette équipe, un merci tout particulier à Yohan Boichut pour sa participation efficace et motivante à la plupart des sujets abordés dans ce document. Et, ça ne va sans doute pas s'arrêter là. Merci également à Axel Legay pour nos discussions préliminaires, avec Benoît Boyer, sur la vérification de propriétés temporelles sur les automates d'arbres. De toutes façons, je sais très bien que s'il n'est pas remercié il va faire la gueule.

Cette longue liste seraient incomplète si elle n'incluait pas Isabelle Puaut qui a gentiment accepté de faire six mois supplémentaires en tant que responsable du master 1 afin de me permettre d'achever, dans de bonnes conditions, ma période de délégation et ce document. Isabelle, je sais que tu étais vraiment prête à tout pour trouver un remplaçant, mais sache que j'apprécie ton geste!

Enfin, je souhaite remercier ma famille pour son soutien indéfectible, même s'ils ne savent pas vraiment ce que je fais et que je pourrais tout aussi bien construire des bagnoles à La Janais. Je remercie de tout coeur Valérie qui m'a soutenu pendant toutes ces années bien qu'elle sache, elle, effectivement à quoi je passe mes journées. Merci aussi à Maxime, Sarah et Lou pour leur énergie communicative et, en même temps, pour leur compréhension, leur patience et leur calme quand papa travaille à la maison.

Contents

1 Preliminaries **9**

2 State of the art **13**
2.1 Term rewriting systems preserving the regularity 13
2.1.1 Known classes of the literature 13
2.1.2 The FPO and GFPO criteria and algorithm 16
2.1.3 The L-IOSLT algorithm 19
2.2 Approximations of reachable terms 19
2.2.1 Equational abstraction 19
2.2.2 Inferring equational abstractions on tree automata 21
2.3 Other analysis with tree abstract domains 21
2.3.1 Analysis of tree transducers 21
2.3.2 Analysis of imperative, functional and logic programs using regular languages 24

3 Theoretical contributions **31**
3.1 The standard Tree Automata Completion algorithm 31
3.1.1 The standard critical pair solving 32
3.1.2 Normalization rules 37
3.1.3 The exact case 37
3.1.4 Extension to the conditional case 45
3.2 Equational Tree Automata Completion 49
3.2.1 Simplification of Tree Automata by Equations 49
3.2.2 $\mathcal{R}/E$-coherent Tree Automata 53
3.2.3 One step of equational completion 58
3.2.4 The full Completion Algorithm 63
3.3 Comparison with other work 68
3.3.1 Recognizing regular classes with standard completion 68
3.3.2 Recognizing regular classes with equational completion 72
3.3.3 Equational abstraction 74
3.3.4 Regular and abstract regular tree model-checking 76
3.3.5 Static Analysis 77

4 Practical contributions **83**
4.1 The Timbuk library . . . 83
4.1.1 History . . . 83
4.1.2 Timbuk tree automata completion and reachability analysis tool . . 85
4.1.3 Taml . . . 93
4.1.4 Tabi . . . 96
4.1.5 Known users . . . 98
4.2 Tree automata matching optimization . . . 99
4.2.1 Basic matching algorithm . . . 99
4.2.2 An optimized algorithm for epsilon-free automata . . . 101
4.2.3 Dealing with epsilon transitions and matching . . . 104
4.3 Implementation and use of matching in Timbuk . . . 106
4.3.1 Tabling and propagation of new substitutions . . . 106
4.3.2 Simplification of tree automata with equations . . . 108
4.4 Tree automata completion extensions . . . 108
4.4.1 Discussion about non left-linear term rewriting systems . . . 108
4.4.2 Restricted conditional term rewriting systems . . . 112
4.5 Tree automata completion checker . . . 114
4.6 Comparison with other tools . . . 115
4.6.1 On regular classes . . . 116
4.6.2 Other tree automata completion tools . . . 118
4.6.3 Completion as an alternative for breadth-first search . . . 120
4.6.4 Equational abstraction with Maude . . . 122

5 Applications **127**
5.1 Cryptographic protocol verification . . . 127
5.1.1 The problem . . . 127
5.1.2 Encoding into TRS and verification by tree automata completion . . 129
5.1.3 The SmartRight case study . . . 131
5.1.4 Comparison with other tools . . . 136
5.2 Prototyping of static analyzers . . . 136
5.2.1 Translation of Java Bytecode Semantics into TRS . . . 138
5.2.2 Analysis in the single threaded case . . . 144
5.2.3 Analysis of multi-threaded programs . . . 147
5.2.4 Copster . . . 149
5.3 Proof of strong non termination . . . 150

6 Conclusion **153**

Bibliography **157**

Introduction

This work is concerned with software verification techniques, i.e. how to prove properties on softwares. The techniques we focus on do not require to execute the program. These techniques are commonly qualified as *static* in opposition to *dynamic* techniques like testing that require to run the program. Static techniques prove properties on a model of the program built either from its code or from its specification. *Model-checking* proves a temporal property on a program specification that is a specific model of a program. This technique is based on the exploration of the graph of reachable program states which needs to be finite or, at least, have a finite description. *Static analysis* builds an abstraction of the program from its source code and then proves properties on this model. These models, obtained using *abstract interpretation*, permit to prove properties on infinite sets of reachable program states. However, if the abstraction is too coarse, static analysis is not abble to prove the expected property. Besides to these techniques, the complete specification and verification can be performed using a *proof assistant*. In that case, the logic embedded in the proof assistant is usually expressive enough to specify the program and prove the expected property. However, the proof is generally done by hand with few automatic steps.

In the domain of static analysis and model-checking, verification techniques have to be fully automatic. Hence, a verification technique needing additional information, provided by a human, to succeed is not considered as reasonable. On the opposite, in the domain of user-assisted proof, a verification technique that fails to prove a property and that cannot be guided so as to finish one particular proof is frustrating. Our objective is to propose a verification technique in between. On the one side, the technique we propose is based on abstractions so as to handle infinite models. On the other side, when abstractions are too coarse and proof fails, our technique permits to guide the verification by hand using approximation refinement.

A verification framework based on rewriting

For the verification technique to be as general purpose as possible, the formalism needs to be simple and expressive enough for modeling heterogeneous software systems. This is the reason why we use Term Rewriting Systems (TRS for short) to model programs. TRS are used for automated deduction for more than 40 years. More recently, they have been used for program modeling. TRS are particularly well suited for this purpose and they can model in a simple and readable way a large variety of computational systems. For instance, TRS can model deduction systems, functions, parallel processes or transition systems for

which rewriting respectively models: deduction, evaluation, progression or transitions. In addition, rewriting makes it also possible to model any combination of them, for instance: two processes executing functional programs or, as shown in Section 5.2.1, several threads executing Java programs. Finally, another interest of the rewriting framework is that it can be used to model programs either at high level of abstraction, e.g. a specification, or at low level, e.g. machine code. Cryptographic protocols are an example of a high level specification modeled in an abstract way using TRS. Messages are represented using terms and dynamics of the protocol using TRS. Rewriting models the reaction of each agent when receiving a specific message, i.e. rewrite rules are of the form "on reception of this message, I send this message". Though it is simple, this model is enough to perform very deep and efficient verifications (Armando et al., 2005; Genet and Klay, 2000) and to come up with tricky attacks on classical cryptographic protocols of the literature (Chevalier and Vigneron, 2002) and even on industrial protocols under development (Heen et al., 2008). By opposition to those high level models, we also use rewriting to model the execution of Java at the lowest level, i.e. at the bytecode level (Boichut et al., 2007). In this case, terms encode the Java Virtual Machine (JVM) state and rewriting represents the execution of each bytecode instruction of the program.

The core verification technique: proving (un)-reachability on rewriting

In the field of rewriting, the reachability problem is well-known: given a term rewriting system $\mathcal{R}$ and two ground terms s and t, t is said to be $\mathcal{R}$-reachable from the *initial* term s (denoted by $s \rightarrow_{\mathcal{R}}{}^* t$) if s can be rewritten into t with a finite number of $\mathcal{R}$ rewriting steps. On the opposition, t is $\mathcal{R}$-unreachable from s (denoted by $s \not\rightarrow_{\mathcal{R}}{}^* t$) if there exists no way to rewrite s into t.

Reachability and unreachability proofs can be used as general purpose verification techniques for the systems modeled using rewriting. For deduction systems, functions, parallel processes, transition systems, proving unreachability will respectively show that a property cannot be deduced, a function call cannot be evaluated into a forbidden value, that a critical configuration (like a deadlock) will never happen, or that a state is unreachable from the initial configuration. On the two above examples: cryptographic protocols and Java bytecode, an unreachability proof can ensure that a security protocol does not leak a secret or that the execution of a given Java program never puts the JVM in a forbidden state.

All the contributions presented in this document are related to software verification using (un)reachability proofs on term rewriting systems. When the TRS $\mathcal{R}$ is terminating and the set of initial terms s is finite, the problem is decidable. Indeed, to decide if $s \rightarrow_{\mathcal{R}}{}^* t$ it is enough to rewrite s by $\mathcal{R}$, in all possible ways, and check if it can be rewritten into t. Since $\mathcal{R}$ is terminating, the rewriting tree is finite and so is its exploration. This is close to a naive model-checking algorithm using exploration of the model. The situation is different when $\mathcal{R}$ is not terminating or when the set of initial term is not finite. For instance, TRSs used for the verification of security protocols are not terminating (See Section 5.1). One of the reasons for that is that the TRS encodes an unbounded number of successive

protocol sessions. For Java program verification, though the TRS encoding the execution of a program may terminate, it is uneasy to show it because it is huge. Besides to this, the objective is generally to prove the safety of the program for any set of entries, i.e. an unbounded number of initial terms s.

In those two situations, a finite exploration of the rewriting tree is impossible. However, under some syntactic restrictions on $\mathcal{R}$ (detailed in Section 2.1.1), for a given set of initial terms $\mathcal{L}$, the *set of reachable terms* $\mathcal{R}^*(\mathcal{L}) = \{t \mid s \in \mathcal{L} \ \wedge \ s \rightarrow_{\mathcal{R}}{}^* \ t\}$ can be computed. Since $\mathcal{R}$ may not be terminating and $\mathcal{L}$ may not be finite, $\mathcal{R}^*(\mathcal{L})$ is not necessarily finite. Infinite sets of terms are usually finitely represented using *tree automata*. Thus, given a tree automaton recognizing the set $\mathcal{L}$ and for some classes of TRS $\mathcal{R}$, the set $\mathcal{R}^*(\mathcal{L})$ is recognized by a tree automaton $\mathcal{B}$ and the reachability problem becomes decidable. Indeed, deciding if $\mathcal{B}$ recognizes t is easy and it is equivalent to deciding if $t \in \mathcal{R}^*(\mathcal{L})$.

Nevertheless, as we will see in Section 2.1.1, syntactical classes for which there exists an automaton recognizing exactly $\mathcal{R}^*(\mathcal{L})$ are very restrictive and few TRSs modeling real programs fall into those classes. For general TRS, it is possible to build an over-approximation $Approx$ of $\mathcal{R}^*(\mathcal{L})$ and thus give a criterion for unreachability only: $s \not\rightarrow_{\mathcal{R}}{}^* \ t$ if $s \in \mathcal{L}$ and $t \notin App$.

Contributions

This document surveys our main contributions to the proof of (un)reachability on term rewriting systems as well as two applications. Our first contribution concerns the precision of the tree automata completion algorithm proposed in (Genet, 1998) whose role is to build an automaton $Appro$ over-approximating $\mathcal{R}^*(\mathcal{L})$. We show that, under *exact normalizing strategy*, this algorithm computes exactly the set of reachable terms for many interesting TRS classes. This is the case for most of the linear TRS classes for which the set of reachable terms is known to be regular: Ground TRS (Dauchet and Tison, 1990; Brainerd, 1969), Linear and Semi-Monadic (Coquidé et al., 1991), Linear and (inversely) Growing (Jacquemard, 1996), Linear Generalized Semi-Monadic (Gyenizse and Vágvölgyi, 1998), Linear Finite-Path Overlapping (Takai et al., 2000), Linear Generalized Finite-Path Overlapping (Takai, 2004), Constructor Based (Réty, 1999). As a corollary, our results provide simpler proofs of regularity for those classes. Furthermore, since the tree automata completion algorithm was initially designed to build over-approximations, the *same* algorithm can do both. Thus, tree automata completion computes reachable terms exactly when the TRS belongs to one of the above classes and permits to over-approximate them otherwise.

The second contribution is to have proposed and integrated in the completion procedure two languages for defining approximations: normalization rules and approximation equations. Normalization rules are an ad-hoc language that mimics the structure of the completed tree automaton. As a result, this language lets the user fully adapt the approximation to its verification objective. However, since it is a low-level language, the precision of the approximations defined using normalization rules is difficult to estimate. This is the reason why we propose another language, approximation equations, whose semantics is formal and is based on equivalence classes. Using this language makes it possible to prove

a second precision result which is that any automaton produced by completion with a set of approximation equations recognizes no more than terms reachable by rewriting modulo the set of equations.

While the above results tend to refine the precision of completion and the precision of approximations, we also worked on extending its scope. A third contribution is to have extended the tree automata completion to the case of conditional term rewriting systems. Using conditions significantly improves the expressiveness of the language used to model the systems to verify and can lead to shorter specifications.

The fourth contribution is the implementation of all the above results in a tool. This tool, Timbuk, permits to compute exactly the automaton recognizing the set of reachable terms when possible and to build an over-approximation using normalization rules or equations, otherwise. This part is particularly concerned with the optimization of several of the algorithms used in tree automata completion.

The last contribution, is to have applied reachability analysis using Timbuk on realistic verification problems, namely verification of cryptographic protocol and static analysis of Java bytecode programs. For cryptographic protocols, our work was one of the first to over-approximate the execution of a protocol in an unbounded Dolev-Yao model, i.e. unbounded number of protocol sessions, unbounded number of agents and unbounded number of symbolic intruder actions. This approach was then refined so as to deal with an industrial copy-protection protocol designed by Thomson R&D. For Java bytecode verification, our objective was to show that tree automata completion is an interesting way of fast prototyping and fine tuning static analysis to a particular property to prove. Finally, since all the above analysis rely on a similar model, i.e. a completed tree automaton, they can all be certified using a *common* fixpoint checker. In particular, this checker is independent of the program to verify, the property to prove and the precision of the approximation.

Outline of the document

Chapter 1 gives some definitions and notations necessary to read the document. Then, in Chapter 2, we present a survey of known classes of TRS for which the set of reachable terms $\mathcal{R}^*(\mathcal{L})$ is known to be computable exactly when $\mathcal{L}$ is recognized by a tree automaton. In the same chapter, we also survey works closely related to over-approximation of reachable terms. In Chapter 3, we propose our algorithm called *tree automata completion* as well as the two languages we use to define approximations: normalization rules and equations. In this chapter, we show that completion without any approximation, computes exactly $\mathcal{R}^*(\mathcal{L})$ for left-linear TRSs falling into the decidable classes of Section 2.1.1. Given a set of approximation equations E, we show a precision result saying that completion computes no more than $\mathcal{R}/E$-reachable terms, i.e. terms that can be reached by rewriting initial terms with $\mathcal{R}$ modulo E. At the end of the chapter, we draw some comparison with other techniques for exact and approximated computations of $\mathcal{R}^*(\mathcal{L})$ presented in Chapter 2. Chapter 4, presents Timbuk as well as some other tools designed to experiment with tree automata and tree automata completion. Finally, Chapter 5 details our two main case studies of reachability analysis on TRS using tree automata completion, namely cryptographic protocol verification and Java bytecode verification.

Chapter 1

Preliminaries

Comprehensive surveys can be found in (Dershowitz and Jouannaud, 1990; Baader and Nipkow, 1998) for term rewriting systems, and in (Comon et al., 2008; Gilleron and Tison, 1995) for tree automata and tree language theory.

Definition 1 (Terms, $\mathcal{T}(\mathcal{F})$ and $\mathcal{T}(\mathcal{F},\mathcal{X})$) *Let $\mathcal{F}$ be a finite set of symbols, each associated with an arity, and let $\mathcal{X}$ be a countable set of variables. $\mathcal{T}(\mathcal{F},\mathcal{X})$ denotes the set of terms, and $\mathcal{T}(\mathcal{F})$ denotes the set of ground terms (terms without variables).* $\diamond$

Definition 2 (Term variables, $\mathcal{V}ar(t)$) *The set of variables of a term t is denoted by $\mathcal{V}ar(t)$.* $\diamond$

Definition 3 (Substitution) *A substitution is a function σ from $\mathcal{X}$ into $\mathcal{T}(\mathcal{F},\mathcal{X})$, which can be extended uniquely to an endomorphism of $\mathcal{T}(\mathcal{F},\mathcal{X})$.* $\diamond$

Definition 4 (Term position) *A position p for a term t is a word over $\mathbb{N}$. The empty sequence ϵ denotes the top-most position. The set $\mathcal{P}os(t)$ of positions of a term t is inductively defined by:*

- $\mathcal{P}os(t) = \{\epsilon\}$ *if* $t \in \mathcal{X}$
- $\mathcal{P}os(f(t_1,\ldots,t_n)) = \{\epsilon\} \cup \{i.p \mid 1 \leq i \leq n$ *and* $p \in \mathcal{P}os(t_i)\}$.

$\diamond$

Definition 5 (Subterm at position p, $t|_p$) *If $p \in \mathcal{P}os(t)$, then $t|_p$ denotes the subterm of t at position p.* $\diamond$

Definition 6 (Term replacement, $t[s]_p$) *$t[s]_p$ denotes the term obtained by replacement of the subterm $t|_p$ at position p by the term s.* $\diamond$

Definition 7 (Multiple term replacement, $t[p_i \leftarrow t_i \mid 1 \leq i \leq m]$) *Let t be a term and $p_1,\ldots,p_m$ be positions of t such that p_i and p_j are disjoint if $i \neq j$. The replacement in t of subterms of t at positions $p_1,\ldots,p_m$ with terms $t_1,\ldots,t_m$ is denoted by $t[p_i \leftarrow t_i \mid 1 \leq i \leq m]$* $\diamond$

Definition 8 (Ground context, $C[\,]$) *A ground context $C[\,]$ is a term of $\mathcal{T}(\mathcal{F} \cup \{\Box\})$ containing exactly one occurrence of the symbol $\Box$. If $t \in \mathcal{T}(\mathcal{F})$ then $C[t]$ denotes the term obtained by the replacement of $\Box$ by t in $C[\,]$.* ◇

Definition 9 (Rewrite rule) *A rewriting rule is a pair of terms $(l, r) \in \mathcal{T}(\mathcal{F}, \mathcal{X}) \times \mathcal{T}(\mathcal{F}, \mathcal{X})$, denoted by $l \to r$, where $l \notin \mathcal{X}$, and $\mathcal{V}ar(l) \supseteq \mathcal{V}ar(r)$.* ◇

Definition 10 (Term Rewriting System, TRS) *A term rewriting system $\mathcal{R}$ is a set of rewrite rules.* ◇

Definition 11 (Equation) *An equation is a pair of terms $(s, t) \in \mathcal{T}(\mathcal{F}, \mathcal{X}) \times \mathcal{T}(\mathcal{F}, \mathcal{X})$, denoted by $s = t$.* ◇

Definition 12 (Linear terms) *A term t is linear if each variable of $\mathcal{V}ar(t)$ occurs only once in t.* ◇

Definition 13 (Linear, Left-linear, Right-linear rewrite rule and TRS) *A rewrite rule $l \to r$ is* left-linear *(resp.* right-linear*) if l (resp. r) is linear. A TRS is left-linear (resp. right-linear), if all its rewrite rules are left-linear (resp. right-linear). A rewrite rule (or a TRS) is* linear *if it is left and right-linear.* ◇

Definition 14 (Linear equation and linear set of equations) *An equation $s = t$ is linear if s and t are linear. A set of equations is linear if all its equations are linear.* ◇

Definition 15 (Rewrite relation, $\to_{\mathcal{R}}$) *Let $\mathcal{R}$ be a TRS and $l \to r$ a rewrite rule of $\mathcal{R}$. Let $s, t \in \mathcal{T}(\mathcal{F})$. The term s can be rewritten into t, denoted by $s \to_{\mathcal{R}} t$, if there exists a position $p \in \mathcal{P}os(s)$ and a substitution σ such that $s|_p = l\sigma$ and $t = s[r\sigma]_p$, i.e.*

$$s = s[l\sigma]_p \to_{\mathcal{R}} s[r\sigma]_p = t$$

◇

The reflexive transitive closure of $\to_{\mathcal{R}}$ is denoted by $\to_{\mathcal{R}}^*$.

Definition 16 (Termination of $\mathcal{R}$, $\to_{\mathcal{R}}$) *A term rewriting system $\mathcal{R}$, or the associated rewrite relation $\to_{\mathcal{R}}$, is said to be terminating if there is no infinite rewrite chain $s \to_{\mathcal{R}} s_1 \to_{\mathcal{R}} \ldots$* ◇

Definition 17 (Normal form, irreducible term, $IRR(\mathcal{R})$) *A term $s \in \mathcal{T}(\mathcal{F}, \mathcal{X})$ is said to be in normal form or irreducible w.r.t. a TRS $\mathcal{R}$ if there exists no term $t \in \mathcal{T}(\mathcal{F}, \mathcal{X})$ such that $s \to_{\mathcal{R}} t$. The set of terms irreducible w.r.t. $\mathcal{R}$ is denoted by $IRR(\mathcal{R})$.* ◇

Definition 18 (Rewriting up to a normal form, ${\to_{\mathcal{R}}}^!$) *A term $s \in \mathcal{T}(\mathcal{F}, \mathcal{X})$ can be rewritten into a normal form $t \in \mathcal{T}(\mathcal{F}, \mathcal{X})$ if $s \to_{\mathcal{R}}^* t$ and $t \in IRR(\mathcal{R})$. This is denoted by $s \to_{\mathcal{R}}^! t$.* ◇

Definition 19 (Confluence of $\mathcal{R}$, $\rightarrow_{\mathcal{R}}$) *A term rewriting system $\mathcal{R}$, or the associated rewrite relation $\rightarrow_{\mathcal{R}}$, is said to be confluent if for all terms $s, t, u \in \mathcal{T}(\mathcal{F}, \mathcal{X})$ such that*

$$\begin{tikzcd} & s \arrow[dl, "\mathcal{R}"', "*"] \arrow[dr, "\mathcal{R}", "*"'] & \\ t & & u \end{tikzcd}$$

then there exists a term $v \in \mathcal{T}(\mathcal{F}, \mathcal{X})$ such that

$$\begin{tikzcd} t \arrow[dr, "*", "\mathcal{R}"'] & & u \arrow[dl, "*"', "\mathcal{R}"] \\ & v & \end{tikzcd}$$

◇

Definition 20 (E-equivalence or equality modulo E) *The E-*equivalence *relation is defined as follows.*

- *First, for two ground terms $t, t' \in \mathcal{T}(\mathcal{F})$ and an equation $l = r$, we say that $t = t'$ if there exists a substitution $\tau : \mathcal{X} \mapsto \mathcal{T}(\mathcal{F})$ such that $l\tau = t$ and $r\tau = t'$;*
- *Then, we define the equivalence relation $=_E \subseteq \mathcal{T}(\mathcal{F}) \times \mathcal{T}(\mathcal{F})$ as the smallest congruence containing the relation $\{(t, t') \in \mathcal{T}(\mathcal{F}) \times \mathcal{T}(\mathcal{F}) \mid t = t'\}$.*

◇

Definition 21 (E-equivalence class or quotient of a set of terms by a set of equations E) *The* quotient of the set $\mathcal{L} \subseteq \mathcal{T}(\mathcal{F})$ by a set of equations E, *denoted by $\mathcal{L}/E$, is the set of sets of terms defined as follows.*

- *For $t \in \mathcal{T}(\mathcal{F})$, $[t]_E$ denotes the $=_E$-equivalence class of t;*
- *Then, $\mathcal{L}/E = \{[t]_E \mid t \in \mathcal{L}\}$.*

◇

Definition 22 ($\overleftrightarrow{E}$) *For any set of equations E, $\overleftrightarrow{E} = \{l \rightarrow r, r \rightarrow l \mid l = r \in E\}$.* ◇

Definition 23 ($\mathcal{R}$-descendants, $\mathcal{R}^*(\mathcal{L})$) *The set of $\mathcal{R}$-descendants of a term set $\mathcal{L} \subseteq \mathcal{T}(\mathcal{F})$ is $\mathcal{R}^*(\mathcal{L}) = \{t \in \mathcal{T}(\mathcal{F}) \mid \exists s \in \mathcal{L}$ s.t. $s \rightarrow_{\mathcal{R}}^* t\}$. We extend this notation to terms in the following way: $\mathcal{R}^*(s) = \mathcal{R}^*(\{s\})$.* ◇

Definition 24 ($\mathcal{R}$-normal forms, $\mathcal{R}^!(\mathcal{L})$) *The set of $\mathcal{R}$-normal forms of a term set $\mathcal{L} \subseteq \mathcal{T}(\mathcal{F})$ is $\mathcal{R}^!(\mathcal{L}) = \mathcal{R}^*(\mathcal{L}) \cap IRR(\mathcal{R})$.* ◇

Definition 25 (Equational rewriting, rewriting modulo a set of equations) *Given a TRS $\mathcal{R}$, a set of equations E and two terms $s, t \in \mathcal{T}(\mathcal{F})$, $s \rightarrow_{\mathcal{R}/E} t \iff \exists s', t' \in \mathcal{T}(\mathcal{F})$ s.t. $s =_E s' \rightarrow_{\mathcal{R}} t' =_E t$.* ◇

The relation $\rightarrow^*_{\mathcal{R}/E}$ is the reflexive transitive closure of $\rightarrow_{\mathcal{R}/E}$.

Definition 26 ($\mathcal{R}/E$-descendants) *The set of $\mathcal{R}/E$-descendants of a language of terms $\mathcal{L} \subseteq \mathcal{T}(\mathcal{F})$ is $\mathcal{R}^*_E(\mathcal{L}) = \{t \in \mathcal{T}(\mathcal{F}) \mid \exists s \in \mathcal{L}$ s.t. $s \rightarrow^*_{\mathcal{R}/E} t\}$.* ◇

The verification technique we propose in this paper is based on the computation of $\mathcal{R}^*(\mathcal{L})$. Note that $\mathcal{R}^*(\mathcal{L})$ is possibly infinite: $\mathcal{R}$ may not terminate and/or $\mathcal{L}$ may be infinite. The set $\mathcal{R}^*(\mathcal{L})$ is generally not computable (Gilleron and Tison, 1995). However, it is possible to over-approximate it (Feuillade et al., 2004; Takai, 2004) using tree automata, i.e. a finite representation of infinite (regular) sets of terms. We next define tree automata.

Let $\mathcal{Q}$ be a finite set of symbols, with arity 0, called *states* such that $\mathcal{Q} \cap \mathcal{F} = \emptyset$. $\mathcal{T}(\mathcal{F} \cup \mathcal{Q})$ is called the set of *configurations*.

Definition 27 (Transition, normalized transitions and epsilon transitions) *A* transition *is a rewrite rule $c \rightarrow q$, where c is a configuration i.e. $c \in \mathcal{T}(\mathcal{F} \cup \mathcal{Q})$ and $q \in \mathcal{Q}$. A* normalized transition *is a transition $c \rightarrow q$ where $c = f(q_1, \ldots, q_n)$, $f \in \mathcal{F}$ an n-ary symbol, and $q_1, \ldots, q_n \in \mathcal{Q}$. An* epsilon transition *$c \rightarrow q$ is such that $c \in \mathcal{Q}$.* ◇

Definition 28 (Bottom-up non-deterministic finite tree automaton (NFTA)) *A bottom-up non-deterministic finite tree automaton (tree automaton for short) is a quadruple $\mathcal{A} = \langle \mathcal{F}, \mathcal{Q}, \mathcal{Q}_f, \Delta \rangle$, where $\mathcal{Q}_f \subseteq \mathcal{Q}$ and Δ is a set of normalized transitions and epsilon transitions.* ◇

The *rewriting relation* on $\mathcal{T}(\mathcal{F} \cup \mathcal{Q})$ induced by the transitions of $\mathcal{A}$ (the set Δ) is denoted by $\rightarrow_\Delta$. When Δ is clear from the context, $\rightarrow_\Delta$ will also be denoted by $\rightarrow_{\mathcal{A}}$.

Definition 29 (Recognized language) *The tree language recognized by $\mathcal{A}$ in a state q is $\mathcal{L}(\mathcal{A}, q) = \{t \in \mathcal{T}(\mathcal{F}) \mid t \rightarrow^*_{\mathcal{A}} q\}$. The language recognized by $\mathcal{A}$ is $\mathcal{L}(\mathcal{A}) = \bigcup_{q \in \mathcal{Q}_f} \mathcal{L}(\mathcal{A}, q)$. A tree language is regular if and only if it can be recognized by a tree automaton.* ◇

Example 30 (Tree automaton and recognized language) *Let $\mathcal{F} = \{f, g, a\}$ and $\mathcal{A} = \langle \mathcal{F}, \mathcal{Q}, \mathcal{Q}_f, \Delta \rangle$, where $\mathcal{Q} = \{q_0, q_1, q_2, q_3, q_4\}$, $\mathcal{Q}_f = \{q_0\}$, and $\Delta = \{f(q_1) \rightarrow q_0, g(q_1, q_1) \rightarrow q_1, a \rightarrow q_1\}$. The languages recognized by q_1 and q_0 are the following: $\mathcal{L}(\mathcal{A}, q_1)$ is the set of terms built on $\{g, a\}$, i.e. $\mathcal{L}(\mathcal{A}, q_1) = \mathcal{T}(\{g, a\})$, and $\mathcal{L}(\mathcal{A}, q_0) = \mathcal{L}(\mathcal{A}) = \{f(x) \mid x \in \mathcal{L}(\mathcal{A}, q_1)\}$.*

We also define epsilon and epsilon-free derivations which are necessary both for completion and simplification of tree automaton by equation application.

Definition 31 (epsilon and epsilon-free derivations) *We denote by $\xrightarrow{\epsilon}$ rewritings performed by epsilon transitions. Conversely we denote by $s \rightarrow^{\not\epsilon}_{\mathcal{A}} t$ (resp. $s \rightarrow^{\not\epsilon}_{\Delta} t$) the fact that $s \rightarrow_{\mathcal{A}} t$ (resp. $s \rightarrow_\Delta t$) and no epsilon transition has been used for rewriting.* ◇

Chapter 2

State of the art

In this chapter, we review some of the papers where regular languages are used to compute or over-approximate the semantics of a TRS, tree transducer or more generally of a program. The first section is devoted to the computation of the image of a regular language by a TRS. We first review the known classes of TRS for which the image of a regular language is regular. These classes are also known as *classes of TRS preserving the regularity*. Then, we also review some papers dealing with regular over-approximations of the image. In Section 2.3, we present some other papers not dealing with TRS but which are also using regular languages to model the semantics of tree transducers, functional programs and imperative programs.

2.1 Term rewriting systems preserving the regularity

2.1.1 Known classes of the literature

The basic reachability problem we are going to consider is the following: given a term rewriting system $\mathcal{R}$ and two terms $s, t \in \mathcal{T}(\mathcal{F})$, can we decide whether $s \rightarrow_{\mathcal{R}}^{\star} t$ or not? In this part, we focus on the existing solutions designed for particular cases. The simplest case is when $\mathcal{R}$ is terminating. Here is a simple but inefficient procedure: to decide whether $s \rightarrow_{\mathcal{R}}{}^{\star} t$ or not it is enough to see if $t \in \mathcal{R}^*(s)$ since $\mathcal{R}^*(s)$ is finite and computable.

When $\mathcal{R}$ is not terminating, deciding reachability needs some additional formal tools. For instance, tree automata can be used to finitely represent the infinite set $\mathcal{R}^*(s)$ and then check if $t \in \mathcal{R}^*(s)$. Many works are devoted to the construction of $\mathcal{R}^*(\mathcal{S})$ for a regular language $\mathcal{S}$ and a term rewriting system $\mathcal{R}$. The set $\mathcal{R}^*(\mathcal{S})$ is clearly not always regular, choose for instance $\mathcal{S} = \{f(a, b)\}$ and $\mathcal{R} = \{f(x, y) \rightarrow f(s(x), s(y))$. In this case, $\mathcal{R}^*(\mathcal{S}) = \{f(s^n(a), s^n(b)) \mid n \in \mathbb{N}\}$. In fact, it was shown in (Gilleron and Tison, 1995) that deciding whether $\mathcal{R}^*(\mathcal{S})$ was regular is not possible in general, even if $\mathcal{R}$ is a confluent and terminating linear TRS. Thus, most of the results define classes of TRS $\mathcal{R}$ so that $\mathcal{R}^*(\mathcal{S})$ is regular. First, we present the classes which can be defined with simple syntactic restrictions on $\mathcal{R}$:

G: $\mathcal{R}$ is a ground TRS (Dauchet and Tison, 1990; Brainerd, 1969).

RL-M: $\mathcal{R}$ is a right-linear and monadic TRS (Salomaa, 1988), i.e. right-hand sides of the rules of $\mathcal{R}$ are either variables or terms of the form $f(x_1, \ldots, x_n)$ where $f \in \mathcal{F}$ and $x_1, \ldots, x_n$ are variables.

L-SM: $\mathcal{R}$ is a linear and semi-monadic TRS (Coquidé et al., 1991), i.e. rules are linear and their right-hand sides are of the form $f(t_1, \ldots, t_n)$ where $f \in \mathcal{F}$ and $\forall i = 1, \ldots, n$, t_i is either a variable or a ground term.

L-G^{-1}: In (Jacquemard, 1996), Jacquemard defines the class **L-G** of linear "growing" TRS, where "growing" means that every left-hand side is either a variable, or a term $f(t_1, \ldots, t_n)$ where $f \in \mathcal{F}$, $\mathcal{A}r(f) = n$, and for all $i = 1, \ldots, n$ the term t_i is a variable, a ground term, or a term whose variables do not occur in the right-hand side. Jacquemard also shows that if $\mathcal{R}$ is growing then $(\mathcal{R}^{-1})^*(\mathcal{S})$ was regular. Those classes are essentially used for neededness analysis of redex in rewriting, see for instance (Durand and Middeldorp, 1997). In order to compare this class with all the others on $\mathcal{R}^*(\mathcal{S})$, we can define the **L-G^{-1}** class using the restrictions in the other direction. The **L-G^{-1}** class corresponds to linear TRS where the *right-hand side* is either a variable, or a term $f(t_1, \ldots, t_n)$ where $f \in \mathcal{F}$, $\mathcal{A}r(f) = n$, and for all $i = 1, \ldots, n$ the term t_i is a variable, a ground term, or a term whose variables do not occur in the *left-hand side*. Thus, note that in this class the usual variable restriction on rewrite rules, i.e. $\mathcal{V}ar(l) \supseteq \mathcal{V}ar(r)$ does not hold.

RL-G^{-1}: similar to **L-G^{-1}** except that left-linearity is not required. This result was proved by Nagaya and Toyama in (Nagaya and Toyama, 1999).

L-IOSLT: $\mathcal{R}$ is a linear I/O separated layered transducing TRS (Seki et al., 2002). Those TRS are defined on sets of symbols $\mathcal{F}_i$, F_o and P such that $\forall p \in P' : \mathcal{A}r(g) = 1$ and $\mathcal{F}_i$, $\mathcal{F}_o$ and P are disjoint. Symbols of $\mathcal{F}_i$ are input symbols and those of $\mathcal{F}_o$ are output symbols. In the TRS, all the rewrite rules are of the form:

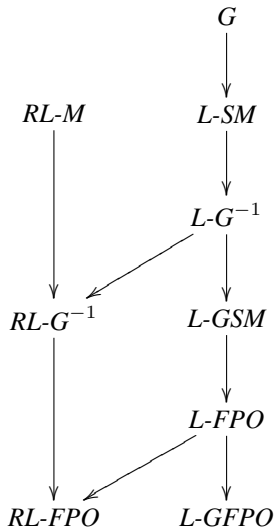
- $f_i(p_1(x_1), \ldots, p_n(x_n)) \to p(t_o)$, or
- $p'_1(x_1) \to p'(t'_o)$

where $f_i \in \mathcal{F}_i$, $p_1, \ldots, p_n, p, p'_1, p' \in P$, $x_1, \ldots, x_n$ are disjoint variables, $t_o, t'_o \in \mathcal{T}(\mathcal{F}_o, \mathcal{X})$ such that $\mathcal{V}ar(t_o) \subseteq \{x_1, \ldots, x_n\}$ and $\mathcal{V}ar(t'_o) \subseteq \{x_1\}$. This class corresponds to linear tree transducers as explained in section 2.3.1.

Recently, new and more general classes were found. The classes of **L-GSM** linear generalized semi-monadic TRS (Gyenizse and Vágvölgyi, 1998), **RL-FPO** right-linear and finite-path overlapping TRS, **L-FPO** linear finite-path overlapping TRS (Takai et al., 2000) and **L-GFPO** linear generalized finite-path overlapping TRS (Takai, 2004). The regularity criteria used in classes **L-GSM**, **RL-FPO**, **L-FPO** and **L-GFPO** are more sophisticated and cannot be expressed as a simple syntactic restriction like above classes. They are based on a careful inspection of the syntactic structure of rewrite rules so that recursive application of rewrite rules are guaranteed to preserve regularity. We give some details on the **RL-FPO** and **RL-GFPO** criteria in Section 2.1.2. Thanks to some results of (Takai et al., 2000) and

in (Takai, 2004), the expressiveness of all the above mentioned classes can be ordered in the following way.

Proposition 32 (Expressiveness of TRS classes)



and L-IOSLT is incomparable with others.

P. Réty (Réty, 1999) proposed another way of considering the problem and defined a class where restrictions are weaker on the TRS and stronger on the regular language $\mathcal{S}$. Since this class imposes restrictions on the language $\mathcal{S}$ it is thus incomparable with previous ones. We call this class **constructor based**. The alphabet $\mathcal{F}$ is separated into a set of *defined symbols* $\mathcal{D} = \{f \mid \exists l \to r \in \mathcal{R} \text{ s.t. } \mathcal{R}oot(l) = f\}$ and constructor symbols $\mathcal{C} = \mathcal{F} \setminus \mathcal{D}$. The restriction on $\mathcal{S}$ is the following: $\mathcal{S}$ is the set of ground constructor instances of a linear term t, i.e. $\mathcal{S} = \{t\sigma\}$ where $t \in \mathcal{T}(\mathcal{F}, \mathcal{X})$ is linear and $\sigma : \mathcal{X} \mapsto \mathcal{T}(\mathcal{C})$. The restrictions on $\mathcal{R}$ are the following: for each rule $l \to r$

1. r is linear, and

2. for each position $p \in \mathcal{P}os_{\mathcal{F}}(r)$ such that $r|_p = f(t_1, \ldots, t_n)$ and $f \in \mathcal{D}$ we have that for all $i = 1 \ldots n$, t_i is a variable or a ground term, and

3. there is no nested function symbols in r

Finally, some works also consider the construction of sets of reachable terms under classical evaluation strategies of rewriting or modulo theory. Réty and Vuotto have shown in (Réty and Vuotto, 2002) that $\mathcal{R}^*(\mathcal{S})$ is still regular, with the same restriction as (Réty, 1999) on $\mathcal{R}$ and $\mathcal{S}$, when rules of $\mathcal{R}$ are applied under some specific strategies: innermost, outermost, ... In (Bouajjani and Touili, 2005), Bouajjani and Touili show that the set of reachable terms for subclasses of Process Rewrite Systems (Mayr, 1998) can be computed exactly. Process Rewrite Systems can be seen as a particular case of TRS modulo the associativity, commutativity and neutrality of some symbols.

2.1.2 The FPO and GFPO criteria and algorithm

As seen above, **RL-FPO** and **L-GPFO** are some of the most general classes of TRS $\mathcal{R}$ such that $\mathcal{R}^*(\mathcal{S})$ is regular if $\mathcal{S}$ is. We now detail the FPO and GFPO criteria which are based on the notion of *sticking-out*. Roughly a term s sticks out of a term t if they have in common a position p such that (1) all symbols encountered on the path from ϵ to p are the same in s and t, and (2) $t|_p$ is a variable and $s|_p$ is either a variable or a non ground term. This is formally defined as follows (Takai et al., 2000).

Definition 33 *A term s* sticks-out *of a term t at position p with $p \in \mathcal{P}os_{\mathcal{X}}(t) \setminus \{\epsilon\}$ if*

- $\forall o : \epsilon \preceq o \prec p \wedge o \in \mathcal{P}os(s) \implies s(o) = t(o)$, *and*
- $p \in \mathcal{P}os(s)$ *and* $s|_p \notin \mathcal{T}(\mathcal{F})$.

Additionally, a term s properly sticks-out *of a term t at position p it $s|_p$ is not a variable.*
◇

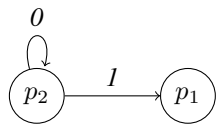
Example 34 *Assume that $\mathcal{F} = \{f, g, a, b\}$ and $\mathcal{X} = \{x, y, z, u\}$. The term $f(x, g(b))$ sticks out of term $f(y, a)$ at position $1.\epsilon$ and $f(f(a, x), y)$ properly sticks-out of $f(z, g(u))$ at position $1.\epsilon$.*

Definition 35 (Sticking-out graph) *The* sticking-out graph *of a TRS $\mathcal{R}$ is a directed graph $G = (V, E)$ where $V = \mathcal{R}$, the vertices are the rules of $\mathcal{R}$, and the set E is defined as follows. Let v_1 and v_2 be, possibly identical, vertices which corresponds to rewrite rules $l_1 \to r_1$ and $l_2 \to r_2$ respectively. For i=1,2, replace each variable in $\mathcal{V}ar(r_i) \setminus \mathcal{V}ar(l_i)$ by a fresh constant symbol ♦.*

1. *If r_2 properly sticks-out of a subterm of l_1, then E contains an edge from v_2 to v_1 with weight one.*
2. *If a subterm of r_2 properly sticks-out of l_1, then E contains an edge from v_2 to v_1 with weight one.*
3. *If a subterm of l_1 sticks-out of r_2, then E contains an edge from v_2 to v_1 with weight zero.*
4. *If l_1 sticks-out of a subterm of r_2, then E contains an edge from v_2 to v_1 with weight zero.*

◇

Example 36 *Let $\mathcal{F} = \{f, g, a, b\}$, $\mathcal{X} = \{x, y\}$ and $\mathcal{R} = \{p_1 = f(x, a) \to f(h(y), x), p_2 = g(y) \to f(g(y), b)$ where for $i = 1, 2$ l_i (resp. r_i) denotes the left (resp. right)-hand side of p_i. Since y occurs in r_1 but not in l_1 it is replaced by the ♦ constant. Since $r_2 = f(g(y), b)$ properly sticks out of $f(x, a)$ at position $1.\epsilon$, in E we have an edge between p_2 and p_1 of weight 1. Then, since $l_2 = g(y)$ sticks-out of $r_2 = f(g(y), b)$ at position $1.\epsilon$, in E we have an cyclic edge of weight 0 on p_2. Note that there is no cyclic edge on p_1 because $r_1 = f(h(♦), x)$ does not sticks-out of $l_1 = f(x, a)$ at position $1.\epsilon$ because $r_1|_{1.\epsilon} = h(♦)$ which is a ground term. The complete sticking-out graph is thus the following:*



Definition 37 *A TRS is* Finite Path Overlapping *(FPO) if its sticking-out graph has no cycle of weight* 1 *or more.* ◇

Theorem 38 (RL-FPO TRS preserve regularity (Takai et al., 2000)) *If $\mathcal{S}$ is a regular language $\mathcal{S}$ and $\mathcal{R}$ is a right-linear FPO (RL-FPO) TRS then $\mathcal{R}^*(\mathcal{S})$ is regular.*

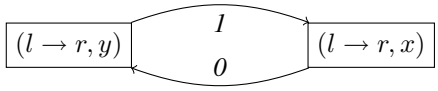
When dealing with linear TRS, the criterion can be improved as shown in (Takai, 2004). It is based on the notion of *Generalized sticking-out graph* we now define.

Definition 39 (Generalized sticking-out graph) *The* generalized sticking-out graph *of a TRS $\mathcal{R}$ is a directed graph $G = (V, E)$. The set V of vertices is defined by $V = \{(l \to r, x) \mid l \to r \in \mathcal{R}$ and $x \in \mathcal{V}ar(l) \cup \mathcal{V}ar(r)\}$. The set E of edges is defined as follows. Let $l_1 \to r_1$ and $l_2 \to r_2$ be two rules of $\mathcal{R}$, possibly identical.*

1. *If r_2 properly sticks-out at position p of $l_1|_{p'}$ with $p' \in \mathcal{P}os(l_1)$, then for $y = l_1|_{p.p'}$ and for all variables $x \in \mathcal{V}ar(r_2|_p)$, E contains edges from $(l_2 \to r_2, x)$ to $(l_1 \to r_1, y)$ with weight one.*
2. *If $l_1|_{p'}$ with $p' \in \mathcal{P}os(l_1)$ sticks-out of r_2 at position p, then for $x = r_2|_p$ and for all variables $y \in \mathcal{V}ar(l_1|_{p.p'})$, E contains edges from $(l_2 \to r_2, x)$ to $(l_1 \to r_1, y)$ with weight zero.*
3. *If $r_2|_{p'}$ with $p' \in \mathcal{P}os(r_2)$ properly sticks-out of l_1 at position p, then for $y = l_1|_p$ and for all variables $x \in \mathcal{V}ar(r_2|_{p.p'})$, E contains edges from $(l_2 \to r_2, x)$ to $(l_1 \to r_1, y)$ with weight one.*
4. *If l_1 sticks-out of $r_2|_{p'}$ at position p, then for $x = r_2|_{p.p'}$ and for all variables $y \in \mathcal{V}ar(l_1|_p)$, E contains edges from $(l_2 \to r_2, x)$ to $(l_1 \to r_1, y)$ with weight zero.*

◇

Example 40 *Let $\mathcal{R} = \{h(f(x, h(g(y)))) \to f(g(k(y)), h(x))\}$, $l = h(f(x, h(g(y))))$ and $r = f(g(k(y)), h(x))$. With $p' = 1.\epsilon$, we have that $l|_{1.\epsilon} = f(x, h(g(y)))$ (properly) sticks-out of $r = f(g(k(y)), h(x))$ at position $p = 2.1.\epsilon$. We are in the second case of the previous definition. We thus get that there are edges between $r|_p = x$ and all variables of $l|_{p'.p}$ which is the set $\{y\}$. Hence we have an edge between $(l \to r, x)$ and $(l \to r, y)$ of weight* 0*. Symmetrically, still with $p' = 1.\epsilon$, $r = f(g(k(y)), h(x))$ properly sticks-out of $l|_{p'} = f(x, h(g(y)))$ at position $p = 1.\epsilon$. This corresponds to the first case of the previous definition. Hence, there are edges between all variables of $\mathcal{V}ar(r|_p) = \{y\}$ and $x = l|_{p'.p}$. Thus, there is an edge between $(l \to r, y)$ and $(l \to r, x)$ of weight* 1*. Here is the complete generalized sticking-out graph:*



Definition 41 *A TRS is* Generalized Finite Path Overlapping *(GFPO) if its generalized sticking-out graph has no cycle of weight* 1 *or more.* ◇

Theorem 42 (L-GFPO TRS preserve regularity (Takai, 2004)) *If $\mathcal{S}$ is a regular language $\mathcal{S}$ and $\mathcal{R}$ is a linear GFPO (L-GFPO) TRS then $\mathcal{R}^*(\mathcal{S})$ is regular.*

Theorem 43 (Expressiveness of GFPO (Takai, 2004)) L-GFPO $\supset$ L-FPO

We now present the algorithm used to construct the automaton recognizing $\mathcal{R}^*(\mathcal{S})$ when $\mathcal{R}$ is in $L - GFPO$. The algorithms proposed in (Takai et al., 2000) and (Takai, 2004) are very similar. In fact, on linear TRS they are exactly the same. We chose to detail the one for $L - GFPO$ because it is simpler and shows the main common idea of both algorithms which is the notion of *packed state* (named structured state in (Takai, 2004)).

Definition 44 (Packed state (Takai et al., 2000) and Packed automaton) *For a set of symbols $\mathcal{F}$ and a set of states $\mathcal{Q}$, the set of packed states, denoted by $\mathcal{P}_{\mathcal{F},\mathcal{Q}}$, is defined by:*

- *if $q \in \mathcal{Q}$ then $\{q\} \in \mathcal{P}_{\mathcal{F},\mathcal{Q}}$, and*
- *if $\mathcal{F} \in \mathcal{F}$, $\mathcal{A}r(f) = n$ and $p_1, \ldots, p_n \in \mathcal{P}_{\mathcal{F},\mathcal{Q}}$ then $\{f(p_1, \ldots, p_n)\} \in \mathcal{P}_{\mathcal{F},\mathcal{Q}}$, and*
- *if $p_1, p_2 \in \mathcal{P}_{\mathcal{F},\mathcal{Q}}$ then $p_1 \cup p_2 \in \mathcal{P}_{\mathcal{F},\mathcal{Q}}$.*

If $\mathcal{A}$ is a tree automaton, the packed automaton $pack(\mathcal{A})$ *is the tree automaton where, for all state $q \in \mathcal{A}$, all occurrences of q in $\mathcal{Q}, Q_f, \Delta$ are replaced by $\{q\}$.* ◇

For readability, a packed state $\{p_1, \ldots, p_n\}$ is written as $\langle p_1, \ldots, p_n\rangle$.

Example 45 *Let $\mathcal{F} = \{f, g\}$ and $\mathcal{Q} = \{q_1, q_2\}$. Here are some possible packed states of $\mathcal{P}_{\mathcal{F},\mathcal{Q}}$: $\langle q_1\rangle$, $\langle q_1, q_2\rangle$, $\langle f(g(\langle q_2\rangle), \langle q_1, q_2\rangle)\rangle$.*

Then, the tree automata construction can be defined as follows using the procedure addtrans and modify.

Procedure addtrans(t) This procedure takes a term t of $\mathcal{T}(\mathcal{F} \cup \mathcal{Q})$ as input and adds new packed states to $\mathcal{Q}$ and new transitions to Δ.

- if $\langle t\rangle \in \mathcal{Q}$ then do nothing;
- if t is a constant then add state $\langle t\rangle$ to $\mathcal{Q}$ and transition $t \to \langle t\rangle$ to Δ;
- if $t = f(t_1, \ldots, t_n)$ then add transition $f(\langle t_1\rangle, \ldots, \langle t_n\rangle) \to \langle t\rangle$ and execute addtrans(t_i) for all $1 \leq i \leq n$.

Example 46 *Assume that $\mathcal{Q} = \{\langle q\rangle\}$ and $\Delta = \{a \to \langle q\rangle\}$. If we call* addtrans*($f(\langle q\rangle, f(a, b))$), $\mathcal{Q}$ becomes $\{\langle q\rangle, \langle a\rangle, \langle b\rangle, \langle f(\langle a\rangle, \langle b\rangle)\rangle, \langle f(\langle q\rangle, \langle f(\langle a\rangle, \langle b\rangle)\rangle)\rangle\}$ and $\Delta = \{a \to \langle q\rangle, a \to \langle a\rangle, b \to \langle b\rangle, f(\langle a\rangle, \langle b\rangle) \to \langle f(\langle a\rangle, \langle b\rangle)\rangle, f(\langle q\rangle, \langle f(\langle a\rangle, \langle b\rangle)) \to \langle f(\langle q\rangle, \langle f(\langle a\rangle, \langle b\rangle)\rangle)\rangle\}$.*

Procedure $\mathsf{modify}(\mathcal{A}, \mathcal{R})$

Input: tree automaton $\mathcal{A} = \langle \mathcal{F}, \mathcal{Q}, \mathcal{Q}_f, \Delta \rangle$ and a linear TRS $\mathcal{R}$.

Output: a tree automaton $\mathcal{A}'$ defined as follows.

For any rule $l \to r \in \mathcal{R}$, any substitution $\sigma : \mathcal{X} \mapsto \mathcal{Q}$ and any packed state q such that $l\sigma \to_{\mathcal{A}}{}^* q$ and $r\sigma \not\to_{\mathcal{A}}{}^* q$ then $\mathcal{A}'$ is obtained from $\mathcal{A}$ by adding the transition $\langle r\sigma \rangle \to q$ to $\mathcal{A}$ and running $\mathsf{addtrans}(r\sigma)$.

Theorem 47 (Fixpoint of modify **(Takai, 2004))** *If there is a fixpoint automaton $\mathcal{A}_*$ for the equation $X = \mathsf{modify}(X, \mathcal{R}) \cup \mathcal{A}$ then $\mathcal{R}^*(\mathcal{L}(\mathcal{A})) = \mathcal{L}(\mathcal{A}_*)$.*

Note that this is very close to the tree automata completion algorithm defined in Sections 3.1 and Section 3.2.

2.1.3 The L-IOSLT algorithm

For the **L-IOSLT** class, we can define the algorithm using similar techniques, though it is not exactly the case in (Seki et al., 2002). We use again the packed states defined in the previous section and we define the following $\mathsf{modifySLT}$ procedure.

Procedure $\mathsf{modifySLT}(\mathcal{A}, \mathcal{R})$

Input: tree automaton $\mathcal{A} = \langle \mathcal{F}, \mathcal{Q}, \mathcal{Q}_f, \Delta \rangle$ and a linear **L-IOSLT** TRS $\mathcal{R}$.

Output: a tree automaton $\mathcal{A}'$ defined as follows.

For all substitution $\sigma : \mathcal{X} \mapsto \mathcal{Q}$, for all rewrite rule $l \to r \in \mathcal{R}$ and all packed state $q \in \mathcal{Q}$ such that $l\sigma \to_{\mathcal{A}}{}^* q$ and $r\sigma \not\to_{\mathcal{A}}{}^* q$:

- if $l = f(p_1(x_1), \ldots, p_n(x_1)) \to p(x_i) = r$ with $(1 \leq i \leq n)$ then $\mathcal{A}'$ is obtained from $\mathcal{A}$ by adding $p(x_i\sigma) \to q$ to $\mathcal{A}$;
- $l = p'_1(x) \to p'(x) = r$ then $\mathcal{A}'$ is obtained from $\mathcal{A}$ by adding $p'(x\sigma) \to q$ to $\mathcal{A}$;
- if $l = f(p_1(x_1), \ldots, p_n(x_1)) \to p(g(t_1, \ldots, t_n)) = r$ then $\mathcal{A}'$ is obtained from $\mathcal{A}$ by adding the transitions $g(\langle t_1\sigma \rangle, \ldots \langle t_n\sigma \rangle) \to [q, p]$ and $p([q, p]) \to q$ to $\mathcal{A}$ where $[q, p]$ is a state made of the pair q and p., Then, we run $\mathsf{addtrans}(t_1\sigma)$, ..., $\mathsf{addtrans}(t_n\sigma)$;
- $l = p'_1(x) \to p'(g(t_1, \ldots, t_n)) = r$ then $\mathcal{A}'$ is obtained from $\mathcal{A}$ by adding the transitions $g(\langle t_1\sigma \rangle, \ldots \langle t_n\sigma \rangle) \to [q, p']$ and $p'([q, p']) \to q$ to $\mathcal{A}$ and running $\mathsf{addtrans}(t_1\sigma)$, ..., $\mathsf{addtrans}(t_n\sigma)$

Theorem 48 (Fixpoint of $\mathsf{modifySLT}$ **(Seki et al., 2002))** *If $\mathcal{R}$ is a* **L-IOSLT** *TRS then there is a fixpoint automaton $\mathcal{A}_*$ for the equation $X = \mathsf{modifySLT}(X, \mathcal{R}) \cup \mathcal{A}$ and $\mathcal{R}^*(\mathcal{L}(\mathcal{A})) = \mathcal{L}(\mathcal{A}_*)$.*

2.2 Approximations of reachable terms

2.2.1 Equational abstraction

The papers (Meseguer et al., 2003, 2008) are not, strictly speaking, about computing a regular language approximating the semantics of a TRS $\mathcal{R}$. However, the approximations

defined using, so-called, equational abstraction have much in common. In their framework, TRS are used to represent a transition relation between states encoded by terms. Properties are defined in Linear Temporal Logic (LTL) and verified using a model-checker that is implemented, using the rewriting logic, in Maude (Clavel et al., 2001). When the system has an infinite set of states, they propose to use a set E of equations to define equivalence classes of states. If this set E is well chosen, then the set of equivalence classes is finite and the corresponding approximation preserves the properties to prove. They show that it can help in proving safety properties, i.e. something bad will not happen. Their work go even further and show that, even with an over-approximation, it is possible to prove liveness properties, i.e. something good will finitely happen.

The framework Meseguer et al. defined is very generic and powerful. Conjointly to TRS, it also uses equations to state the axioms of the model. For simplicity, we chose here to focus on a restricted part of their framework where the system is *only* defined using a TRS $\mathcal{R}$ (with no axioms). Moreover, we only consider simple safety properties encoded by $s \not\rightarrow_{\mathcal{R}}{}^* t$ where s and t are terms encoding, respectively, an initial state and a bad state. This is simpler than their framework, but the core problem is still there: since $\mathcal{R}$ may rewrite infinitely s, how prove that $s \not\rightarrow_{\mathcal{R}}{}^* t$? They propose to define a set of equations E, show that $s \not\rightarrow^*_{\mathcal{R}/E} t$ and use the fact that $\rightarrow_{\mathcal{R}} \subseteq \rightarrow_{\mathcal{R}/E}$ to prove that $s \not\rightarrow_{\mathcal{R}}{}^* t$. Under certain conditions, even if $\mathcal{R}$ infinitely rewrites s, $s \not\rightarrow^*_{\mathcal{R}/E} t$ may be finitely proven. This is the case if the set $\mathcal{R}^*_E(s)$ contains only a finite set of E-equivalence classes and no one contains t.

However, proving $s \not\rightarrow^*_{\mathcal{R}/E} t$ using only rewriting and a tool like Maude is not easy. Indeed, for rewriting with $\mathcal{R}/E$ the equational part of E is generally encoded using rewriting. More precisely, E is oriented into another TRS $\mathcal{R}'$ terminating and ground confluent. If $\mathcal{R}'$ enjoys those two properties, equality modulo E becomes decidable. For two ground terms s and t, $s =_E t$ iff there exists a term u such that $s \rightarrow^!_{\mathcal{R}'} u$ and $t \rightarrow^!_{\mathcal{R}'} u$. Let $\rightarrow_{\mathcal{R}(\mathcal{R}')}$ be the relation $\rightarrow_{\mathcal{R}(\mathcal{R}')} = \rightarrow^*_{\mathcal{R}'} \circ \rightarrow_{\mathcal{R}} \circ \rightarrow^*_{\mathcal{R}'}$. Since $\mathcal{R}'$ has been obtained by orienting equations of E, if $s \rightarrow^*_{\mathcal{R}(\mathcal{R}')} t$ then $s \rightarrow^*_{\mathcal{R}/E} t$. However the implication does not always hold in the opposite direction.

Example 49 ((Viry, 2002)) *Let $\mathcal{R} = \{f(x+y) \rightarrow g(x) + g(y)\}$ and $E = \{x + 0 = x\}$. Equations of E can be oriented into $\mathcal{R}' = \{x + 0 \rightarrow x\}$ which is terminating and ground confluent. We can prove that $f(0+0) + 0 \rightarrow^*_{\mathcal{R}/E} g(0) + g(0)$ because $f(0+0) + 0 \rightarrow_{\mathcal{R}} g(0) + g(0) + 0 \rightarrow_{\mathcal{R}'} g(0) + g(0)$. However, $f(x)$ cannot be rewritten by $\rightarrow_{\mathcal{R}(\mathcal{R}')}$ though $f(x) \rightarrow_{\mathcal{R}/E} g(x) + g(0)$ since $f(x) =_E f(x+0) \rightarrow_{\mathcal{R}} g(x) + g(0)$.*

Having $\rightarrow_{\mathcal{R}(\mathcal{R}')} \subseteq \rightarrow_{\mathcal{R}/E}$ and not $\rightarrow_{\mathcal{R}(\mathcal{R}')} \supseteq \rightarrow_{\mathcal{R}/E}$ is a problem for verifying safety properties using equational abstractions. Indeed, the only provable facts with a rewriting tool like Maude are of the form $s \not\rightarrow^*_{\mathcal{R}(\mathcal{R}')} t$ that do not always entail $s \not\rightarrow^*_{\mathcal{R}/E} t$. For this inclusion to be true in the opposite direction, the coherence (Viry, 2002) property has to be proven on $\mathcal{R}$ and $\mathcal{R}'$. Roughly, $\mathcal{R}$ and $\mathcal{R}'$ are coherent if for all term s that can be rewritten into t_1 by $\mathcal{R}$, on one side, and into t_2 by $\mathcal{R}'$, on the other side, then t_1 can be rewritten by $\mathcal{R}'$ and t_2 by $\mathcal{R}$ into a common term u.

$$\begin{array}{ccc} s & \xrightarrow[\mathcal{R}]{} & t_1 \\ \mathcal{R}' \downarrow & & \downarrow *\ \mathcal{R}' \\ t_2 & \xrightarrow[*]{\mathcal{R}} & u \end{array}$$

This property cannot be shown in general but there exists a syntactic condition on $\mathcal{R}$ and $\mathcal{R}'$, called local coherence, that implies coherence. This criterion is based on a standard critical pair technique between the rules of $\mathcal{R}$ and those $\mathcal{R}'$. In (Meseguer et al., 2003, 2008), several examples show that finding a set of approximation equations E that can be oriented into a TRS $\mathcal{R}'$ enjoying termination, ground confluence and coherence with $\mathcal{R}$ is not an easy task. However, when it exists, it permits to prove both safety and liveness properties using only rewriting, and thus, in a very efficient way.

2.2.2 Inferring equational abstractions on tree automata

Additionally to the **L-GFPO** class (see Section 2.1.1), (Takai, 2004) proposes over-approximate the set of reachable terms when it is not regular. It is defined as an extension of its modify procedure. The extension proposed is based on abstract interpretation (Cousot and Cousot, 1977) and consists to add a widening operation to modify. The widening automatically infers approximation from the Generalized sticking-out graph (Section 2.1.2). The inferred widening can be seen as the application of an equation of the form $C[C[x]] = C[x]$ where $C[\,]$ can be any context. The algorithm automatically construct the widening. However, the form of the equation is fixed and cannot, thus, guarantee the existence of a fixpoint for modify for any left-linear TRS.

In particular, equations of the form $C_1[C_2[x]] = C_2[C_1[x]]$ or $C[x] = x$ cannot be found automatically nor manually added. When executing several steps of modify, if a widening position is found, i.e. there exists at least one repetition of a context $C[\,]$, then the widening adds an epsilon transition encoding the repetition of $C[\,]$. This transition directly widens the language of the form $C[C[\ldots]]$ by $C^*[\ldots]$. More precisely, assume that the tree automaton $\mathcal{A}$ is such that $a \rightarrow_{\mathcal{A}}{}^* q_1$, $C[q_1] \rightarrow_{\mathcal{A}}{}^* q_2$ and that $C[q_2] \rightarrow_{\mathcal{A}}{}^* q_3$. We thus have $C[C[q_2]] \rightarrow_{\mathcal{A}}{}^* q_2$. The widening detects the regularity and simply add the epsilon transition $q_3 \rightarrow q_2$. We thus obtain the looping derivation $C[q_2] \rightarrow_{\mathcal{A}}{}^* q_3 \rightarrow q_2$ recognizing any language of the form $C[\ldots C[a]\ldots]$ into q_2.

2.3 Other analysis with tree abstract domains

2.3.1 Analysis of tree transducers

Tree transducers can be viewed as particular cases of TRS where rewriting is applied either bottom-up or top-down. Here, we consider linear bottom-up tree transducer (Gécseg and Steinby, 1984; Comon et al., 2008) that are defined as follows.

Definition 50 (Linear Tree Transducer) *Let $\mathcal{F}_i$ be a set of input symbols, $\mathcal{F}_o$ a set of output symbols and $\mathcal{Q}$ a set of (unary) states such that $\mathcal{F}_i \cap \mathcal{Q} = \emptyset$, $\mathcal{F}_o \cap \mathcal{Q} = \emptyset$ and*

$\forall q \in \mathcal{Q} : \mathcal{A}r(q) = 1$. *A* linear tree transducer T *is a tuple* $\langle \mathcal{Q}, \mathcal{F}_i, \mathcal{F}_o, \mathcal{Q}_f, \Delta \rangle$ *where* Δ *is a set of rewrite rules of the form:*

- $f(q_1(x_1), \ldots, q_n(x_n)) \to q(u)$*, or*
- $q'(x_1) \to q''(u')$

where $f \in \mathcal{F}_i$*,* $q_1, \ldots, q_n, q, q', q'' \in \mathcal{Q}$*,* $x_1, \ldots, x_n$ *are disjoint variables,* $u, u' \in \mathcal{T}(\mathcal{F}_o, \mathcal{X})$ *such that* $\mathcal{V}ar(u) \subseteq \{x_1, \ldots, x_n\}$ *and* $\mathcal{V}ar(u') \subseteq \{x_1\}$*.* $\diamond$

A transducer T recognizes a regular relation $R_T = \{(t, t') \in \mathcal{T}(\mathcal{F}_i) \times \mathcal{T}(\mathcal{F}_o) \mid t \to_\Delta^* q(t')$ where $q \in \mathcal{Q}_f\}$. We can define the image of a language by a transducer as follows: given $\mathcal{S} \subseteq \mathcal{T}(\mathcal{F}_i)$, $R_T(\mathcal{S}) = \{t \in \mathcal{T}(\mathcal{F}_o) \mid s \in \mathcal{S}$ and $(s, t) \in R_T\}$.

Theorem 51 (Recognizability of image (Gécseg and Steinby, 1984; Comon et al., 2008)) *Given a recognizable language* $\mathcal{S}$ *and a linear tree transducer* T*, the set* $R_T(\mathcal{S})$ *is recognizable.*

In (Seki et al., 2002), Seki et al. proposed the class **L-TL** of linear layered transducing TRS which encompasses tree transducers. However, for a **L-TL** TRS $\mathcal{R}$ regularity of $\mathcal{R}^*(\mathcal{S})$ is only ensured for the **L-IOSLT** class defined in Section 2.1.1 and corresponds exactly to linear tree transducers. In the definition of L-IOSLT TRS, $\mathcal{F}_i$ and $\mathcal{F}_o$ are supposed to be disjoint. This is not necessary in tree transducers because rewriting is necessarily performed bottom-up. For instance, if a tree transducer has a rule of the form $f(q(x)) \to q(f(x))$ and thus rewrites $f(q(a))$ into $q(f(a))$ no rewriting can be performed on the subterm $f(a)$. This is not the case in general **L-TL** TRS and this leads to non regular sets of reachable terms like it is shown in Example 5 of (Seki et al., 2002). However, with the restriction $\mathcal{F}_i \cap \mathcal{F}_o$ we cannot have the rewriting rule $f(q(x)) \to q(f(x))$ because f cannot occur both in the left and right-hand side of the rule. This is a simple encoding of the bottom-up strategy used in tree transducers in the symbols used in rewriting.

Several works are considering *Regular Tree Model-Checking* of protocols defined using linear tree transducers. In this setting, the model-checking essentially consists in building the tree automaton representing any number of application of the tree transducer. This problem has been investigated for instance in (Bouajjani and Touili, 2002) and efficient implementations have been proposed in (Abdulla et al., 2005). The construction proposed in (Bouajjani and Touili, 2002) is not limited to tree transducers. They use a widening technique to compute exactly the set of terms reachable by repeated applications of a tree transducer on a regular set of terms. More precisely, if R_T^n is the combination of R_T n times and $\mathcal{A}$ is a tree automaton, the objective is to compute another tree automaton $\mathcal{A}'$ such that $\mathcal{L}(\mathcal{A}') = R_T^*(\mathcal{L}(A)) = \bigcup_{n \geq 0} R_T^n(\mathcal{L}(\mathcal{A}))$, i.e. the reflexive transitive closure of R_T on $\mathcal{L}(\mathcal{A})$. The proposed widening is based on the detection of regularities an the hypergraph encoding of the tree automaton. The first result is that this widening is exact. The second result is that it terminates for a specific class of TRS called Well-Oriented Systems **WOS**. This is due to the fact that the application of TRS of this class can be simulated by the application of a finite number of linear tree transducers. Using repeated applications of Theorem 51, it is thus possible to prove that this class preserve regularity. The **WOS** class can be defined as follows.

Definition 52 (Well-Oriented Systems (Bouajjani and Touili, 2002)) *Let $S = S_0 \cup \ldots \cup S_n$ be disjoint finite sets of symbols. A n-phase well-oriented system over S is a set of rewriting rules of the form:*

$$\begin{array}{rcl}
b(a(x_1,x_2),c_1(x_3,x_4)) & \rightarrow & a(b'(x_1,x_2),c_1(x_3,x_4)) \\
a(b(x_1,x_2),c_1(x_3,x_4)) & \rightarrow & b'(a(x_1,x_2),c_1(x_3,x_4)) \\
a(b(x_1,x_2),c_2(x_3,x_4)) & \rightarrow & b'(a(x_1,x_2),a(x_3,x_4)) \\
b(x_1,x_2) & \rightarrow & d(x_1,x_2) \\
b(a(x_1,x_2),c_1(x_3,x_4)) & \rightarrow & d(a(x_1,x_2),c_1(x_3,x_4)) \\
a(b(x_1,x_2),c_1(x_3,x_4)) & \rightarrow & a(d(x_1,x_2),c_1(x_3,x_4)) \\
b(a(x_1,x_2),c_2(x_3,x_4)) & \rightarrow & a(d(x_1,x_2),d(x_3,x_4))
\end{array}$$

as well as the symmetrical forms of these rules obtained by commuting the children, where $a, b', c_1 \in S_{i+1}$, $b, c_2 \in S_i$, and $d \in S_{i+2}$, such that $0 \leq i \leq n-2$. $\diamond$

The **WOS** class is incomparable with **RL-FPO**, **L-GFPO** and **L-IOSLT** classes. This can be shown as follows:

WOS $\not\subseteq$ RL-FPO: let us construct the sticking-out graph of the first form of rules of **WOS**. Let l_1 (resp. r_1) be the left (resp. right) hand side of the rule. Since r_1 properly sticks-out of the subterm $a(x_1,x_2)$ of l_1, we have a cyclic edge of weight 1 on this rule. Hence it is not **RL-FPO**.

RL-FPO $\not\subseteq$ WOS: non left linear TRS cannot be handled by **WOS**.

WOS $\not\subseteq$ L-GFPO: as above the generalized sticking-out graph of the first form of rules of **WOS** has a cyclic edge of weight 1. Let l_1 (resp. r_1) be the left (resp. right) hand side of the rule. Since r_1 properly sticks-out of the subterm $a(x_1,x_2)$ of l_1, we have a cyclic edge of weight 1 on the node labeled by $(l_1 \rightarrow r_1, x_1)$.

L-GFPO $\not\subseteq$ WOS: collapsing rules, i.e. rules of the form $f(x_1,\ldots,x_n) \rightarrow x_i$ are not covered by **WOS**.

WOS $\not\subseteq$ L-IOSLT: let us have a look at the form of the two first rules of the **WOS** class. In the left-hand sides, a appears at position ϵ in the first rule and at position $1.\epsilon$ in the second one. This is impossible in the **L-IOSLT** (and even in **L-LT**) class where sets of symbols occurring at position ϵ and $i.\epsilon$ in left-hand sides have to be disjoint.

L-IOSLT $\not\subseteq$ WOS: in **L-IOSLT**, we can have rules of the form $q(x) \rightarrow q'(x)$ for all symbols $q, q' \in \mathcal{Q}$. In **WOS**, the only rules of that form are $b(x_1,x_2) \rightarrow d(x_1,x_2)$ where $b \in S_i$, $d \in S_{i+2}$ and $S_i \cap S_{i+2} = \emptyset$. As a result, **L-IOSLT** covers a sets of rules of the form $\{q(x) \rightarrow q'(x), q'(x) \rightarrow q(x)\}$ that are not covered by **WOS**.

When the exact value of $R_T^*(\mathcal{S})$ cannot be finitely computed, the algorithm and widening of (Bouajjani and Touili, 2002) does not terminate. This is why Bouajjani et al. proposed in (Bouajjani et al., 2006a) the *Abstract Regular Tree Model-Checking* where over-approximations of $R_T^*(\mathcal{S})$ are computed, so as to prove some unreachability properties. The paper defines two main abstraction functions which can be seen as equivalence relations on states of the constructed tree automaton. One of the required property is that those

equivalence relations are of finite index, i.e. there is only a finite number of equivalence classes. This property forces the constructed tree automaton to have a finite number of states and thus a finite number of transitions. Hence, the construction of the iterated application of the tree transducer on the tree automaton is forced to terminate. Here are the two proposed equivalence relations:

- two states are equivalent if their recognized languages are equal for terms of height lesser or equal to a given n;
- given a set of tree automata $P = \{P_1, \ldots, P_N\}$, two states q_1 and q_2 of $\mathcal{A}$ are equivalent if $\{P_i \in P \mid \mathcal{L}(P_i) \cap \mathcal{L}(\mathcal{A}, q_1) \neq \emptyset\}$ and $\{P_i \in P \mid \mathcal{L}(P_i) \cap \mathcal{L}(\mathcal{A}, q_2) \neq \emptyset\}$ are equal.

The authors also mention the fact that this last relation has good properties for abstraction refinement. Assume that the approximation relation is too coarse because one of the automata of P, say P_i, recognizes both t_1 and t_2. Roughly, to refine the approximation, it is enough to add two new tree automata P_{N+1} and P_{N+2} recognizing respectively t_1 and t_2 so that the new approximation will distinguish them. For instance, if t_1 is a valid reachable term and t_2 is due to the over-approximation, this is a natural way to exclude t_2 of the approximation. One of the results of (Bouajjani et al., 2006a) is a theorem guaranteeing that any spurious counter-example can be removed using this technique.

2.3.2 Analysis of imperative, functional and logic programs using regular languages

As mentioned in (Cousot and Cousot, 1995), the idea of using regular tree grammar for program analysis is due to (Jones and Muchnick, 1979; Jones, 1987; Jones and Andersen, 2007) following (Reynolds, 1969). Then it has been reformulated by (Heintze and Jaffar, 1990) as *set based analysis*. Note that, set based analysis does not produce regular languages but set constraints. However, the mechanism and the obtained result is very similar to grammars obtained by (Jones, 1987), as it is mentioned in (Heintze, 1993).

We choose here to give a uniform description of the analysis techniques proposed in those papers. Though not all papers actually use them, our equivalent description uses set constraints. The program semantics is encoded into a generic set S of set constraints of the form $x \supseteq f(x_1, \ldots, x_n)$, $x \supseteq y \cup z$, $x \supseteq y \cap z$, etc. defined on a set of variable $X = \{x, y, z, x_1, \ldots\}$. Then, this set of constraints is solved by an algorithm producing a tree grammar over-approximating the solution of S. Again, in all those the papers, the presentation of the solution looks different but its semantics is equivalent. For an imperative program (Cousot and Cousot, 1995), variables of X represent the variables of the program and a set constraint is associated to each program point. For logic programs (Heintze and Jaffar, 1990), each *occurrence* of a variable of the program and each predicate symbol is associated to a variable of X and the clauses of the program are translated into set constraints. In particular, if a variable appears several times in the right-hand side of a clause, i.e. it is non linear, each occurrence is associated to a different variable of X. Finally, for functional programs (Jones and Andersen, 2007), a variable of X is associated to each parameter in function headers and the function definitions are translated into set constraints.

Though the construction of set constraints differs with programming paradigms, their resolution phases are very similar. The only difference is that constraints produced by functional and imperative programs do not contain any constraint of the form $x \supseteq y \cap z$ making the resolution far more easy. This is even mentioned by Heintze in (Heintze, 1993) as the main cause for performance problems of logic program analysis. He even advise to eliminate or limit as much as possible the use of intersection constraints for the modeling of logic programs. In the following, we do not consider intersection constraints.

The approximation used for the resolution of the constraints is similar in all the mentioned papers. The intuition is the following: every set of tuples of the form $\{(a, b), (c, d)\}$ is approximated by the set $\{(a, b), (a, d), (c, b), (c, d)\}$. In other words, the approximation forgets the relations between elements of a tuple in a set. This approximation is called "independent attributes" in (Jones and Andersen, 2007). (Cousot and Cousot, 1995) propose some refinement of this approximation, based on a threshold for targeting the approximation, but the general idea remains the same. Since the results of the resolution techniques are similar, we detail only one of them. We chose the analysis of (Jones, 1987; Jones and Andersen, 2007) because it translates functional programs into TRS and, thus, makes comparison with our techniques on TRS more accurate.

Example 53 *Here is an example of a program, in ML-style, which is needing lazy evaluation to terminate. It is borrowed from (Jones and Andersen, 2007).*

```
let rec first l1 l2=
  match l1 , l2 with
      [ ] , _ -> [ ]
    | l::m, x::xs -> x::(first m xs)

let rec sequence y= y::(sequence (l::y))

let g n= first n (sequence[ ])
```

This program can be encoded into the following TRS $\mathcal{R}$ where 'one' encodes '1', 'cons' encodes '::' and 'nil' encodes the empty list '[]'.

$$first(nil, Xs) \to nil$$
$$first(cons(one, M), cons(X, Xs)) \to cons(X, first(M, Xs))$$
$$sequence(Y) \to cons(Y, sequence(cons(one, Y)))$$
$$g(N) \to first(N, sequence(nil))$$

Note that, the obtained TRS is left-linear by construction because functional definitions are.

The algorithm described in (Jones, 1987; Jones and Andersen, 2007) propose to compute a tree grammar over-approximating the collecting semantics of the function g for a given set of inputs. It can be seen also as an over-approximation of the image of a set of inputs by a function g. Informally, the grammar associates a non-terminal X to each variable X of the program and, for each function call, several non-terminals R_i to represent its set of intermediate results. More precisely, if the function f is defined using k rewrite rules then k non-terminals are associated to f. Note that, when there is no intersection, any

set constraints can be translated into a tree grammar rule and vice versa. For instance, a production of the form $R_0 \underset{G}{\rightarrow} f(a) \mid R_1 \mid R_0$ corresponds to the set constraint $R_0 \supseteq f(a, R_2) \cup R_1 \cup R_0$ with set of variables $\{R_0, R_1, R_2\}$. A first step for the analysis of the program consists in building the grammar of the initial calls for which we want to analyze the program. On the previous TRS, a grammar G describing some interesting inputs can be:

$$\begin{array}{lcl} R_0 & \underset{G}{\rightarrow} & g(\Omega) \\ \Omega & \underset{G}{\rightarrow} & nil \mid cons(Atom, \Omega) \\ Atom & \underset{G}{\rightarrow} & zero \mid one \mid \ldots \end{array}$$

Then, the algorithm tries to match left-hand sides of rules of $\mathcal{R}$ with (subterms of) the right-hand sides of the productions of G. For instance, the definition of $g(N)$ can be matched on $g(\Omega)$ with solution $\{N \mapsto \Omega\}$. In that case several productions and non-terminal are added to the grammar. First, since there is only one rewrite rule defining g, only one non-terminal, say R_1, is associated to any call of g. In our case, R_1 represents the set of intermediate results of the computation of $g(N)$ with $N = \Omega$. Then, one non-terminal per parameter in the definition of g is added, i.e. here only one N. Finally the following productions relating the non-terminal to their values are added.

$$\begin{array}{lcl} R_0 & \underset{G}{\rightarrow} & R_1 \\ R_1 & \underset{G}{\rightarrow} & first(N, sequence(nil)) \\ N & \underset{G}{\rightarrow} & \Omega \end{array}$$

The first production encodes the fact that every result of R_1 is also a result for R_0, the second one unfolds the definition of g and the last explicits the found match between N and Ω. Then, the algorithm continues on G completed with those productions. The next term to be replaced is $sequence(nil)$. Like for g, the $sequence$ function is defined using 1 rewrite rule. Let R_4 be the non-terminal associated to the calls of the $sequence$ function. The left-hand side of the rule of $\mathcal{R}$ defining the $sequence$ function i.e. $sequence(Y)$ can be matched on the subterm $sequence(nil)$ of the above productions leading to a solution $\{Y \mapsto nil\}$.

$$\begin{array}{lcl} R_1 & \underset{G}{\rightarrow} & first(N, R_4) \\ R_4 & \underset{G}{\rightarrow} & cons(Y, sequence(cons(one, Y))) \\ Y & \underset{G}{\rightarrow} & nil \end{array}$$

as before, the first production explicits the link that exists between intermediate results for R_4 and R_1. Then, matching between $sequence(Y)$ and $sequence(cons(one, Y))$ gives the solution $\{Y \mapsto cons(one, Y)\}$. Then, since the $sequence$ function is associated to the unique non-terminal R_4, the term $sequence(cons(one, Y))$ is also replaced by R_4. Note that this naturally leads to looping rules for non-terminal R_4 and Y in the generated productions:

$$\begin{array}{lcl} R_4 & \underset{G}{\rightarrow} & cons(Y, R_4) \\ Y & \underset{G}{\rightarrow} & cons(one, Y) \end{array}$$

Now recall that, in G among all productions, we have the following ones:

$$\begin{array}{lcl} R_1 & \underset{G}{\rightarrow} & first(N, R_4) \\ N & \underset{G}{\rightarrow} & \Omega \\ \Omega & \underset{G}{\rightarrow} & nil \mid cons(Atom, \Omega) \end{array}$$

If we try to match $first(nil, Xs)$ on $first(N, R_4)$, it cannot succeed directly because it is not possible to match nil on N (note that they would they unify). However, since the production $N \underset{G}{\rightarrow}^* nil$ is possible, we can replace N by nil and the problem becomes to match $first(nil, Xs)$ on $first(nil, R_4)$. This succeeds and gives the following solution $\{Xs \mapsto R_4\}$. Since there are two rules defining the $first$ function, let R_2 (resp. R_3) be the non-terminal associated to its first (resp. second) rewrite rule. Since this step only concerns the application of the first rewrite rule of $first$, the algorithm only adds productions for R_2 to G:

$$\begin{array}{lcl} R_1 & \underset{G}{\rightarrow} & R_2 \\ R_2 & \underset{G}{\rightarrow} & nil \\ Xs & \underset{G}{\rightarrow} & R_4 \end{array}$$

Similarly, since in G we find the productions:

$$\begin{array}{lcl} R_1 & \underset{G}{\rightarrow} & first(N, R_4) \\ N & \underset{G}{\rightarrow} & \Omega \\ \Omega & \underset{G}{\rightarrow} & nil \mid cons(Atom, \Omega) \\ R_4 & \underset{G}{\rightarrow} & cons(Y, sequence(cons(one, Y))) \\ R_4 & \underset{G}{\rightarrow} & cons(Y, R_4) \end{array}$$

As before, it is impossible to match $first(cons(one, M), cons(X, Xs))$ on $first(N, R_4)$. However, since $N \underset{G}{\rightarrow}^* cons(one, \Omega)$ on the one side and $R_4 \underset{G}{\rightarrow}^* cons(Y, R_4)$ and $R_4 \underset{G}{\rightarrow} cons(Y, sequence(cons(one, Y)))$ on the other side, this is equivalent to the two problems of matching $first(cons(one, M), cons(X, Xs))$ on $first(cons(one, \Omega), cons(Y, R_4))$ and matching $first(cons(one, M), cons(X, Xs))$ on $first(cons(one, \Omega), cons(Y, sequence(cons(one, Y))))$. Those two matching problems have the following solutions: $\{M \mapsto \Omega, X \mapsto Y, Xs \mapsto R_4\}$ and $\{M \mapsto \Omega, X \mapsto Y, Xs \mapsto sequence(cons(one, Y))\}$, respectively. Note that those two new solutions are obtained for the second rewrite rule of $first$ and thus concerns non-terminal R_3. This results in the following new productions:

$$\begin{array}{lcl} R_1 & \underset{G}{\rightarrow} & R_3 \\ R_3 & \underset{G}{\rightarrow} & cons(X, first(M, Xs)) \\ M & \underset{G}{\rightarrow} & \Omega \\ X & \underset{G}{\rightarrow} & Y \\ Xs & \underset{G}{\rightarrow} & R_4 \mid sequence(cons(one, Y)) \end{array}$$

The last step is achieved by matching $first(cons(one, M), cons(X, Xs))$ on the subterm of the right-hand side of the above second production, i.e. $first(M, Xs)$. Since $M \underset{G}{\rightarrow}^* cons(one, \Omega)$, $Xs \underset{G}{\rightarrow}^* cons(Y, R_4)$ and $Xs \underset{G}{\rightarrow}^* cons(Y, sequence(one, Y))$, as above, we get two solutions for the matching problem: $\{M \mapsto \Omega, X \mapsto Y, Xs \mapsto R_4\}$ and $\{M \mapsto \Omega, X \mapsto Y, Xs \mapsto sequence(one, Y)\}$. Note that this is exactly the same solution

as above, hence all the productions associating the variables to their values are already present in G. Furthermore, since those solutions concern the second rewrite rule of $first$, the subterm $first(M, Xs)$ has to be replaced by R_3. This leads to the following looping transition:

$$R_3 \underset{G}{\rightarrow} cons(X, R_3)$$

Finally, since the left-hand side of the first rewrite rule of $first$, i.e. $first(nil, Xs)$, can be matched on $first(M, Xs)$, this subterm can also be replaced by the non-terminal corresponding to the first rewrite rule: R_2. This adds a last production leading to a stable grammar:

$$R_3 \underset{G}{\rightarrow} cons(X, R_2)$$

Here was a running example of the algorithm of (Jones and Andersen, 2007). Though it is somewhat hidden, the approximation used is exactly the one that is mentioned above: it forgets the relation between elements of a tuple. In fact, it is built in the "one non-terminal by variable" construction principle. For instance, let $f(X, Y)$ be a function and $f(a, b)$, $f(c, d)$ be two function calls. Remember that a unique non-terminal X (resp. Y) is associated to the first (resp. second) parameter of f. Hence, when translating the two calls $f(a, b)$, $f(c, d)$ we obtain the following grammar G:

$$R_0 \underset{G}{\rightarrow} f(X, Y)$$
$$X \underset{G}{\rightarrow} a \mid c$$
$$Y \underset{G}{\rightarrow} b \mid d$$

which is producing the language $\{f(a, b), f(c, b), f(a, d), f(c, d)\}$ where the relational information between $a - b$ and $c - d$ is lost. As mentioned above, the same kind of approximation is used in all the other cited works (Heintze and Jaffar, 1990; Cousot and Cousot, 1995). This is standard because as far as we know, this is the only way to keep an over-approximation *regular* if nothing is known about the program. However, since this approximation is the default behavior of the algorithm and it is built-in, it is difficult to adapt it even if something is known about the program behavior.

The above algorithm can compute an over-approximation of the collecting semantics of a functional program with lazy-evaluation. In fact, it is shown in (Jones and Andersen, 2007) that it also covers *higher-order* functional programs. This is obtained thanks to a specific encoding of higher-order functions into TRS we now present.

Example 54 *Here is a simple higher-order program in ML-style borrowed from (Jones and Andersen, 2007).*

```
let double x = (x,x)
let cons x y= (x,y)

let rec map f l=
  match l with
    [] -> []
  | x::xs -> (f x)::(map f xs)

let f x= (map double x),(map (cons 5) x)
```

This program can be encoded into the following TRS where '`app`*' is an additional symbol encoding function application, '*`a`*' encodes '*5*', '*`fcons`*' encodes '*$cons$*', '*`cons`*' encodes '*$::$*' and '*`nil`*' encodes the empty list* $[]$.

$app(double, X) \to cons(X, X)$
$app(app(fcons, X), Y) \to cons(X, Y)$
$app(app(map, U), nil) \to nil$
$app(app(map, U), cons(X, Xs)) \to cons(app(U, X), app(app(map, U), Xs))$
$app(f, X) \to cons(app(app(map, double), X), app(app(map, app(fcons, a)), X))$

Since the algorithm of Jones and Andersen (2007) is able to deal with any left-linear TRS, having such an encoding is enough to guarantee the feasibility of their analysis on higher-order functions.

Chapter 3

Theoretical contributions

In (Genet, 1998), we proposed the first version of the *tree automaton completion algorithm* for over-approximating $\mathcal{R}^*(\mathcal{S})$ for a left-linear term rewriting system $\mathcal{R}$ and a regular language $\mathcal{S}$. It is referred as the *standard tree automata completion algorithm* in the following and presented in Section 3.1. This algorithm, simple and efficient, was used for the verification of real-sized programs as shown in Chapter 5. This algorithm has been, then, refined into the *equational tree automata completion* (Genet and Rusu, 2009) which is presented in Section 3.2.

3.1 The standard Tree Automata Completion algorithm

Let us first recall the tree automata completion principle. Starting from a tree automaton $\mathcal{A}_0 = \langle \mathcal{F}, \mathcal{Q}, \mathcal{Q}_f, \Delta_0 \rangle$ and a left-linear[1] TRS $\mathcal{R}$, the aim of the approximation algorithm is to compute a tree automaton $\mathcal{A}'$ such that $\mathcal{L}(\mathcal{A}') \supseteq \mathcal{R}^*(\mathcal{L}(\mathcal{A}_0))$. Approximations are used to show that terms recognized by a tree automaton $\mathcal{A}_{bad}$ are not reachable by rewriting terms of $\mathcal{L}(\mathcal{A}_0)$ with $\mathcal{R}$, i.e. $\forall s \in \mathcal{L}(\mathcal{A}_0)\ \forall t \in \mathcal{L}(\mathcal{A}_{bad}) : s \not\rightarrow_{\mathcal{R}}{}^* t$. For this, it is enough to show that $\mathcal{L}(\mathcal{A}') \cap \mathcal{L}(\mathcal{A}_{bad}) = \emptyset$ i.e., compute the automaton recognizing the intersection and show that the recognized language is empty.

The technique consists in successively computing tree automata $\mathcal{A}^1_{\mathcal{R}}, \mathcal{A}^2_{\mathcal{R}}, \ldots$ such that $\forall i \geq 0 : \mathcal{L}(\mathcal{A}^i_{\mathcal{R}}) \subseteq \mathcal{L}(\mathcal{A}^{i+1}_{\mathcal{R}})$ and if $s \in \mathcal{L}(\mathcal{A}^i_{\mathcal{R}})$, such that $s \rightarrow_{\mathcal{R}} t$ then $t \in \mathcal{L}(\mathcal{A}^{i+1}_{\mathcal{R}})$, until we get an automaton $\mathcal{A}^k_{\mathcal{R}}$ with $k \in \mathbb{N}$ such that $\mathcal{L}(\mathcal{A}^k_{\mathcal{R}}) = \mathcal{L}(\mathcal{A}^{k+1}_{\mathcal{R}})$. Thus, $\mathcal{A}^k_{\mathcal{R}}$ is a fixpoint and $\mathcal{A}^k_{\mathcal{R}}$ also verifies $\mathcal{L}(\mathcal{A}^k_{\mathcal{R}}) \supseteq \mathcal{R}^*(\mathcal{L}(\mathcal{A}_0))$. More precisely, to construct $\mathcal{A}^{i+1}_{\mathcal{R}}$ from $\mathcal{A}^i_{\mathcal{R}}$, we achieve a *completion step* which consists in finding *critical pairs* between $\rightarrow_{\mathcal{R}}$ and $\rightarrow_{\mathcal{A}^i_{\mathcal{R}}}$. For a substitution $\sigma : \mathcal{X} \mapsto \mathcal{Q}$ and a rule $l \rightarrow r \in \mathcal{R}$, a critical pair is an instance $l\sigma$ of l such that there exists $q \in \mathcal{Q}$ satisfying $l\sigma \rightarrow^*_{\mathcal{A}^i_{\mathcal{R}}} q$ and $r\sigma \not\rightarrow^*_{\mathcal{A}^i_{\mathcal{R}}} q$. For $r\sigma$ to be recognized by the same state and thus model the rewriting of $l\sigma$ into $r\sigma$, it is enough to add the necessary transitions to $\mathcal{A}^i_{\mathcal{R}}$ to obtain $\mathcal{A}^{i+1}_{\mathcal{R}}$ such that $r\sigma \rightarrow^*_{\mathcal{A}^{i+1}_{\mathcal{R}}}$. The two versions of the tree automata completion differs from the set of added transitions so as

[1] This restriction will be weakened in the following.

to obtain $r\sigma \rightarrow^*_{\mathcal{A}^{i+1}_{\mathcal{R}}}$.

3.1.1 The standard critical pair solving

In the standard tree automata completion, critical pairs are joined in the following way:

$$\begin{tikzcd} l\sigma \arrow[r, "\mathcal{R}"'] \arrow[d, "\mathcal{A}^i_{\mathcal{R}}"', "*"] & r\sigma \arrow[dl, bend left, "*", "\mathcal{A}^{i+1}_{\mathcal{R}}"'] \\ q & \end{tikzcd}$$

Hence, for each critical pair $l\sigma \rightarrow^*_{\mathcal{A}^i_{\mathcal{R}}} q$ and $r\sigma \not\rightarrow^*_{\mathcal{A}^i_{\mathcal{R}}} q$, we add the new transition $r\sigma \rightarrow q$ to $\mathcal{A}^i_{\mathcal{R}}$ to obtain $\mathcal{A}^{i+1}_{\mathcal{R}}$. However, the transition $r\sigma \rightarrow q$ is not necessarily a normalized transition of the form $f(q_1, \ldots, q_n) \rightarrow q'$ and so has to be normalized first. For example, to normalize a transition of the form $f(g(a), h(q')) \rightarrow q$, we need to find some states q_1, q_2, q_3 and replace the previous transition by a set of normalized transitions: $\{a \rightarrow q_1, g(q_1) \rightarrow q_2, h(q') \rightarrow q_3, f(q_2, q_3) \rightarrow q\}$.

Assume that q_1, q_2, q_3 are new states, then adding the transition itself or its normalized form does not make any difference. Now, assume that $q_1 = q_2$, the normalized form becomes $\{a \rightarrow q_1, g(q_1) \rightarrow q_1, h(q') \rightarrow q_3, f(q_1, q_3) \rightarrow q\}$. This set of normalized transitions represents the regular set of non normalized transitions of the form $f(g^*(a), h(q')) \rightarrow q$ which contains the transition we wanted to add initially but also many others. Hence, this is an approximation. We could have made an even more drastic approximation by identifying q_1, q_2, q_3 with q, for instance.

For every transition, there exists an equivalent set of normalized transitions. Normalization consists in decomposing a transition $s \rightarrow q$, into a set $Norm(s \rightarrow q)$ of normalized transitions. In the standard completion algorithm, we abstract subterms s' of s s.t. $s' \notin \mathcal{Q}$ by states of $\mathcal{Q}$. The abstraction function α is defined as follows:

Definition 55 (Abstraction function) *Let $\mathcal{F}$ be a set of symbols, and $\mathcal{Q}$ a set of states. An* abstraction *function α maps every normalized configuration into a state:*

$$\alpha : \{f(q_1, \ldots, q_n) \mid f \in \mathcal{F}, \mathcal{A}r(f) = n \text{ and } q_1, \ldots q_n \in \mathcal{Q}\} \mapsto \mathcal{Q}$$

◇

Definition 56 (Abstraction state) *Let $\mathcal{F}$ be a set of symbols, and $\mathcal{Q}$ a set of states. For a given abstraction function α and for all configuration $t \in \mathcal{T}(\mathcal{F} \cup \mathcal{Q})$ the abstraction state of t, denoted by $top_\alpha(t)$, is defined by:*

1. *if $t \in \mathcal{Q}$, then $top_\alpha(t) = t$,*
2. *if $t = f(t_1, \ldots, t_n)$ then $top_\alpha(t) = \alpha(f(top_\alpha(t_1), \ldots, top_\alpha(t_n)))$.*

◇

Definition 57 (Normalization function) *Let $\mathcal{F}$ be a set of symbols, $\mathcal{Q}$ a set of states, $s \rightarrow q$ a transition s.t. $s \in \mathcal{T}(\mathcal{F} \cup \mathcal{Q})$ and $q \in \mathcal{Q}$, and α an abstraction function. The set $Norm_\alpha(s \rightarrow q)$ of* normalized transitions *is inductively defined by:*

1. *if* $s = q$*, then* $Norm_\alpha(s \to q) = \emptyset$*, and*
2. *if* $s \in \mathcal{Q}$ *and* $s \neq q$*, then* $Norm_\alpha(s \to q) = \{s \to q\}$*, and*
3. *if* $s = f(t_1, \ldots, t_n)$*, then* $Norm_\alpha(s \to q) =$
   $\{f(top_\alpha(t_1), \ldots, top_\alpha(t_n)) \to q\} \cup \bigcup_{i=1}^{n} Norm_\alpha(t_i \to top_\alpha(t_i))$.

◇

Example 58 (Abstraction function and normalization) *Let* $s = f(g(q_1, f(a)))$*, and* α_1 *be the abstraction function* $\{a \mapsto q_4, f(q_4) \mapsto q_3, g(q_1, q_3) \mapsto q_2\}$*. The normalization of transition* $f(g(q_1, f(a))) \to q_0$ *with abstraction* α_1 *is the following:* $Norm_{\alpha_1}(f(g(q_1, f(a))) \to q_0) = \{f(q_2) \to q_0, g(q_1, q_3) \to q_2, f(q_4) \to q_3, a \to q_4\}$*.*

Definition 59 (Regular language substitution) *A* regular language substitution *over an automaton* $\mathcal{A}$ *with a set of states* $\mathcal{Q}$ *is a function* $\sigma : \mathcal{X} \mapsto \mathcal{Q}$*. We can extend this definition to a morphism* $\sigma : \mathcal{T}(\mathcal{F}, \mathcal{X}) \mapsto \mathcal{T}(\mathcal{F}, \mathcal{Q})$*. We denote by* $\Sigma(\mathcal{Q}, \mathcal{X})$ *the set of regular language substitutions built over* $\mathcal{Q}$ *and* $\mathcal{X}$*.* ◇

Definition 60 (One step automaton completion) *Let* $\mathcal{A} = \langle \mathcal{F}, \mathcal{Q}, \mathcal{Q}_f, \Delta \rangle$ *be a tree automaton,* $\mathcal{R}$ *a TRS and* α *an abstraction function. The one step completed automaton* $\mathcal{C}_{\mathcal{R},\alpha}(\mathcal{A})$ *is a tree automaton* $\langle \mathcal{F}, \mathcal{Q}', \mathcal{Q}_f, \Delta' \rangle$ *such that:*

- $\Delta' = \Delta \cup \bigcup_{l \to r \in \mathcal{R},\, q \in \mathcal{Q},\, \sigma \in \Sigma(\mathcal{Q}, \mathcal{X}),\, l\sigma \to^*_\Delta q} Norm_\alpha(r\sigma \to q)$*, and*
- $\mathcal{Q}' = \{q \mid c \to q \in \Delta'\}$

◇

Definition 61 (Automaton completion) *Let* $\mathcal{A}$ *be a tree automaton,* $\mathcal{R}$ *a TRS and* α *an abstraction function.*

- $\mathcal{A}^0_{\mathcal{R},\alpha} = \mathcal{A}$
- $\mathcal{A}^{n+1}_{\mathcal{R},\alpha} = \mathcal{C}_{\mathcal{R},\alpha}(\mathcal{A}^n_{\mathcal{R},\alpha})$ *for* $n \in \mathbb{N}$
- $\mathcal{A}^*_{\mathcal{R},\alpha}$ *is a fixpoint* $\mathcal{A}^*_{\mathcal{R},\alpha} = \mathcal{A}^k_{\mathcal{R},\alpha} = \mathcal{A}^{k+1}_{\mathcal{R},\alpha}$ *with* $k \in \mathbb{N}$

◇

Note that $\mathcal{A}^*_{\mathcal{R},\alpha}$ does not exists in general, but it can be computed in many interesting cases, provided that the abstraction function α ensures termination of the completion. In the following proposition, we give some sufficient conditions for building an over-approximation automaton $\mathcal{B}$ of the set of $\mathcal{R}$-descendants of a regular language recognized by $\mathcal{A}$. First, we need a sufficient condition for the tree automaton $\mathcal{B}$ to over-approximate $({}^*\mathcal{L}(\mathcal{A}))$ when $\mathcal{R}$ is not left-linear.

Definition 62 *Let $\mathcal{A} = \langle \mathcal{F}, \mathcal{Q}, \mathcal{Q}_f, \Delta \rangle$ be an automaton and $\mathcal{R}$ a TRS, $\mathcal{R}$ and $\mathcal{A}$ satisfy the left-coherence condition if:*

$$\forall \tau : \mathcal{X} \mapsto \mathcal{T}(\mathcal{F}),\ \forall l \to r \in \mathcal{R},\ \forall q \in \mathcal{Q} :$$

$$l\tau \to^*_\Delta q \quad \Rightarrow \quad \exists \sigma \in \Sigma(\mathcal{Q}, \mathcal{X}) \ \textit{s.t.}\ l\tau \to^*_\Delta l\sigma \to^*_\Delta q$$

◇

Lemma 63 *Every left-linear TRS $\mathcal{R}$ and every tree automaton $\mathcal{A}$, $\mathcal{R}$ and $\mathcal{A}$ satisfy the left-coherence condition.*

PROOF. In a left-linear TRS, every left-hand side l of a rewrite rule is of the form $l = C[x_1, \ldots, x_n]$ where all the variables of $\{x_1, \ldots, x_n\}$ are distinct and there is no other variable position in C. Since $t = l\tau \to^*_\Delta q$ there exist some terms $t_1, \ldots, t_n$ and some states $q_1, \ldots, q_n$ such that $\tau = \{x_1 \mapsto t_1, \ldots, x_n \mapsto t_n\}$, $t = C[t_1, \ldots, t_n]$ and $\forall i = 1 \ldots n : t_i \to^*_\Delta q_i$. Since all the $x_1, \ldots, x_n$ are distinct, $\sigma = \{x_1 \mapsto q_1, \ldots, x_n \mapsto q_n\}$ is a function and thus is a valid substitution such that $t \to^*_\Delta l\sigma \to^*_\Delta q$.

On the opposite, note that if l is not linear the relation σ we build is not necessarily a function. □

This left-coherence condition is, in fact, necessary for non left-linear TRS. Roughly, the problem with non left-linear rules is the following: let $f(x, x) \to g(x)$ be a rule of $\mathcal{R}$ and let $\mathcal{A}$ be a tree automaton whose set of transitions contains $f(q_1, q_1) \to q_0$ and $f(q_2, q_3) \to q_0$. Although we can construct a valid substitution $\sigma = \{x \mapsto q_1\}$ for matching the rewrite rule on the first transition, it is not the case for the second one. The semantics of a completion between rule $f(x, x) \to g(x)$ and transition $f(q_2, q_3) \to q_0$ would be to find the common language of terms recognized both by q_2 and q_3. This can be obtained by computing a new tree automaton $\mathcal{A}'$ with a set of states $\mathcal{Q}'$ such that $\mathcal{Q}'$ is disjoint from states of $\mathcal{A}$ and $\exists q \in \mathcal{Q}' : \mathcal{L}(\mathcal{A}', q) = \mathcal{L}(\mathcal{A}, q_2) \cap \mathcal{L}(\mathcal{A}, q_3)$. Then, to end the completion step it would be enough to add transitions of $\mathcal{A}'$ to $\mathcal{A}$ with the new transition $g(q) \to q_0$.

On the other hand, one can remark that the non-linearity problem would disappear with deterministic automata since for any deterministic automaton $\mathcal{A}_{det}$ and for all states q, q' of $\mathcal{A}_{det}$ we trivially have $\mathcal{L}(\mathcal{A}, q) \cap \mathcal{L}(\mathcal{A}, q') = \emptyset$. However, determinization of a tree automaton may result in an exponential blow-up of the number of states (Comon et al., 2008).

A solution, between the two previous ones, is to use the *left-coherence condition* defined above by ensuring determinism for a subset of states $q \in \mathcal{Q}$ which are to be matched by the non linear variables of the non linear rules. This is what is called *locally deterministic* tree automata (Genet and Viet Triem Tong, 2001) (see Section 4.4.1).

We now define the condition called *simple left-coherence* which implies the left-coherence condition and which is easier to check on a given tree automaton and TRS. Let $\mathcal{A}$ be an automaton, $l \to r$ a rewrite rule over $\mathcal{T}(\mathcal{F}, \mathcal{X})$, $\{x_1, \ldots, x_k\}$ the set of variables non linear in l and $\mathcal{Y}$ a set of variables distinct from $\mathcal{X}$. Let $Ren(l)$ be the pair (l', E) where l' denotes the term l where non linear variables are renamed and E is a set of constraints.

$$
\begin{array}{lcll}
Ren(l) & = & (l, \emptyset) & \text{if } l \text{ is either a constant or a} \\
 & & & \text{variable that does not} \\
 & & & \text{appear in } \{x_1, \ldots, x_k\} \\
 & = & (y, \{x = y\}) & \text{if } l \text{ is a variable } x \in \{x_1, \ldots, x_k\} \\
 & & & \text{and } y \text{ if a fresh variable of } \mathcal{Y} \\
 & = & (f(t'_1, \ldots, t'_n), \bigcup_{i=1}^{n} E_i) & \text{if } l = f(t_1, \ldots, t_n) \text{ and} \\
 & & & Ren(t_i) = (t'_i, E_i) \text{ for all } i = 1 \ldots n.
\end{array}
$$

Definition 64 (Simple left-coherence condition) *An automaton $\mathcal{A}$ and a TRS $\mathcal{R}$ satisfy the* simplified left-coherence condition *if for all rules $l \to r \in \mathcal{R}$ such that $Ren(l) = (l', E)$:*

$$\forall (x = y) \in E,\ \forall \sigma \in \Sigma(\mathcal{Q}, \mathcal{X}),\ \forall q, q_x, q_y \in \mathcal{Q} :$$

$$l'\sigma \to^*_\Delta q \wedge \sigma(x) = q_x \neq q_y = \sigma(y) \implies \mathcal{L}(\mathcal{A}, q_x) \cap \mathcal{L}(\mathcal{A}, q_y) = \emptyset$$

◇

Proposition 65 *Simple left-coherence implies left-coherence.*

PROOF. If $\mathcal{A}$ does not verify the left-coherence condition induced by $l \to r$, there is at least a ground term t recognized by a state q of $\mathcal{A}$, a substitution $\tau : \mathcal{X} \to \mathcal{T}(\mathcal{F})$ such that $t = l\tau$ and $t \to^*_\Delta q$, and there is no $\mathcal{Q}$-substitution $\sigma \in \Sigma(\mathcal{Q}, \mathcal{X})$ such that $t \to^*_\Delta l\sigma \wedge l\sigma \to^*_\Delta q$. However, if t is an instance of l then t is also instance of l' the renamed version of l, let $t = l'\rho$ where $\rho : \mathcal{Y} \to \mathcal{T}(\mathcal{F})$. The problem is solved by case reasoning on t: t cannot be a variable a otherwise $l = a$ and $\mathcal{A}$ respects the coherence condition, hence t is a term of depth at least 1. Let $\rho : \{y_1, \ldots, y_k\} \mapsto \mathcal{T}(\mathcal{F})$ such that $\rho(y_j) = t_j$. If $t \to^*_\Delta q$ then all subterm of t are recognized by $\mathcal{A}$ and there are k states $q_1, \ldots, q_k$ such that $t_j \to q_j$ for $1 \leq j \leq k$. Observing that $t_j \to q_j$ and $\rho(y_j) = t_j$, we can construct a $\mathcal{Q}$-substitution $\sigma' : \{y_1, \ldots, y_k\} \to \mathcal{Q}$ defined by $\sigma'(y_j) = q_j$. We have $t \to^*_\Delta l'\sigma'$ then $l'\sigma' \to^*_\Delta q$. Either we have found a $\mathcal{Q}$-substitution $\sigma \in \Sigma(\mathcal{Q}, \mathcal{X})$ such that $t \to^*_\Delta l\sigma \wedge l\sigma \to^*_\Delta q$ which contradicts the hypothesis or there are at least two variables y_i and y_j such that

1. $\sigma'(y_i) = q_i$ and $\sigma'(y_j) = q_j$ with $q_i \neq q_j$
2. $y_i = y_j$ is necessarily a constraint of E
3. $t_i = \rho(y_i) = \rho(y_j) = t_j$

The condition 1. and 2. hold true for at least one pair of variables (y_i, y_j) otherwise we could construct a $\mathcal{Q}$-substitution σ such that $l\sigma = l'\sigma'$. The condition 3. holds true because t is an instance of l. In that case, $\mathcal{A}$ does not verify the simple left-coherence which is a contradiction. □

The following proposition states the necessary conditions on two tree automata $\mathcal{A}$ and $\mathcal{B}$ so that the language recognized by $\mathcal{B}$ over-approximate the set of $\mathcal{R}$-descendants of terms recognized by $\mathcal{A}$.

Proposition 66 *Let $\mathcal{R}$ be a TRS, $\mathcal{A} = \langle \mathcal{F}, \mathcal{Q}, \mathcal{Q}_f, \Delta \rangle$, and $\mathcal{B} = \langle \mathcal{F}, \mathcal{Q}', \mathcal{Q}_f, \Delta' \rangle$ two tree automata such that $\mathcal{R}$ and $\mathcal{B}$ satisfy the left-coherence condition. We have $\mathcal{R}^*(\mathcal{L}(\mathcal{A})) \subseteq \mathcal{L}(\mathcal{B})$ if*

1. *$\Delta \subseteq \Delta'$, and*
2. *$\forall l \to r \in \mathcal{R}$, $\forall q \in \mathcal{Q}'$, $\forall \sigma \in \Sigma(\mathcal{Q}', \mathcal{X})$, $l\sigma \to^*_{\Delta'} q$ implies $r\sigma \to^*_{\Delta'} q$.*

PROOF. By definition, any term t of $\mathcal{R}^*(\mathcal{L}(\mathcal{A}))$ is such that $\exists s \in \mathcal{L}(\mathcal{A})$ s.t. $s \to^*_{\mathcal{R}} t$. By induction on the length of the derivation $s \to^*_{\mathcal{R}} t$, we prove that if $s \to^*_{\mathcal{R}} t$ and $s \to^*_{\Delta'} q$ with $q \in \mathcal{Q}_f$ then $t \to^*_{\Delta'} q$, which implies that $t \in \mathcal{L}(\mathcal{B})$.

1. if $t = s$ then, since $s \in \mathcal{L}(\mathcal{A})$, we have that $\exists q \in \mathcal{Q}_f$ s.t. $t = s \to^*_{\Delta} q$. Moreover, $\Delta \subseteq \Delta'$, hence $\exists q \in \mathcal{Q}_f$ s.t. $t \to^*_{\Delta'} q$,
2. if $s \to^+_{\mathcal{R}} t$, then $\exists s' \in \mathcal{T}(\mathcal{F})$ s.t. $s \to^*_{\mathcal{R}} s' \to_{\mathcal{R}} t$. By induction hypothesis applied to $s \to^*_{\mathcal{R}} s'$, we obtain that $\exists q \in \mathcal{Q}_f$ s.t. $s' \to^*_{\Delta'} q$. Moreover, since $s' \to_{\mathcal{R}} t$, there exists a rule $l \to r \in \mathcal{R}$, a substitution τ, and a position p in s' such that $l\tau = s'|_p$ and $t = s'[r\tau]_p$. By construction of bottom-up tree automata with normalized transitions, if $s' \to^*_{\Delta'} q$, then any subterm of s' is reducible by Δ' into a state of $\mathcal{Q}'$. Hence, since $l\tau = s'|_p$, we get that $\exists q' \in \mathcal{Q}'$ s.t. $l\tau \to^*_{\Delta'} q'$ and $s'[q']_p \to^*_{\Delta'} q$. Now, let us show that $r\tau \to^*_{\Delta'} q'$. Let $\mathcal{V}ar(l) = \{x_1, \ldots, x_k\}$ be the variables of l. Since $\mathcal{R}$ and $\mathcal{B}$ satisfy the left-coherence condition, we get that there exists $\sigma \in \Sigma(\mathcal{Q}, \mathcal{X})$ such that $l\tau \to^*_{\Delta'} l\sigma \to^*_{\Delta'} q'$. Hence, there exist some states $q_i \in \mathcal{Q}$ such that $\sigma = \{x_i \mapsto q_i \mid i = 1 \ldots k\}$ and $x_i\tau \to^*_{\Delta'} q_i$ for $i = 1 \ldots k$. From $x_i\tau \to^*_{\Delta'} q_i$ we get that $r\tau \to^*_{\Delta'} r\sigma$. Finally, since $r\sigma \to^*_{\Delta'} q'$, we get that $r\tau \to^*_{\Delta'} q'$ and thus $t = s'[r\tau]_p \to^*_{\Delta'} q$.

□

In this first theorem, we show that completion always over-approximate the set of descendants for TRSs and tree automata satisfying the left-coherence condition.

Theorem 67 *Let $\mathcal{A}$ be a tree automaton, $\mathcal{R}$ be a TRS and α an abstraction function. If $\mathcal{R}$ and $\mathcal{A}^*_{\mathcal{R},\alpha}$ satisfy the left-coherence condition then:*

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

PROOF. For proving $\mathcal{L}(\mathcal{A}^*_{\mathcal{R},\alpha}) \supseteq \mathcal{R}^*\mathcal{L}(A)$, we can use Proposition 66 with $\mathcal{A}$ and $\mathcal{B} = \mathcal{A}^*_{\mathcal{R},\alpha} = \langle \mathcal{F}, \mathcal{Q}', \mathcal{Q}_f, \Delta' \rangle$. Thus, it is enough to prove that the approximation automata $\mathcal{A}^*_{\mathcal{R},\alpha}$ verifies Conditions 1 and 2 of Proposition 66, for all abstraction function α. By Definition 61, $\mathcal{A}^*_{\mathcal{R},\alpha}$ trivially verifies Condition 1. Now, to prove that $\mathcal{A}^*_{\mathcal{R},\alpha}$ also verifies Condition 2 of Proposition 66, it is enough to prove that $Norm_\alpha(r\sigma \to q) \subseteq \Delta'$ implies $r\sigma \to^*_{\Delta'} q$.

Let s' be any subterm of $r\sigma$ (possibly non-strict) and $q' \in \mathcal{Q}'$. By induction on the size of s', we show that $Norm_\alpha(s' \to q') \subseteq \Delta'$ implies that $s' \to^*_{\Delta'} q'$:

- if $s' = q'$, then we trivially have $s' \to^*_{\Delta'} q'$.

- if $s' = q'' \in \mathcal{Q}'$ s.t. $q'' \neq q'$ then, by case 2 of definition of $Norm$, we get that $Norm_\alpha(s' \to q') = \{s' \to q'\}$. Since $Norm_\alpha(s' \to q') \subseteq \Delta'$, we have $s' \to^*_{\Delta'} q'$.
- if $s' = g(t_1, \ldots, t_m) \in \mathcal{T}(\mathcal{F} \cup \mathcal{Q}')$, by applying case 3 of definition of $Norm$, we get that

  (a) $\{g(top_\alpha(t_1), \ldots, top_\alpha(t_m)) \to q'\} \subseteq \Delta'$, and

  (b) $\bigcup_{i=1}^m Norm_\alpha(t_i \to top_\alpha(t_i)) \subseteq \Delta'$,

  where $\forall i = 1 \ldots m$, $top_\alpha(t_i) \in \mathcal{Q}'$ by definition of $\mathcal{A}^*_{\mathcal{R},\alpha}$. By applying the induction hypothesis to (b), we get that $\forall i = 1 \ldots m$, $t_i \to^*_{\Delta'} top_\alpha(t_i)$. On the other hand, (a) implies that $g(top_\alpha(t_1), \ldots, top_\alpha(t_m)) \to_{\Delta'} q'$. As a result, $g(t_1, \ldots, t_m) \to^*_{\Delta'} g(top_\alpha(t_1), \ldots, top_\alpha(t_m)) \to_{\Delta'} q'$.

Hence $Norm_\alpha(r\sigma \to q) \subseteq \Delta'$ implies $r\sigma \to^*_{\Delta'} q$, and Condition 2 of Proposition 66 is satisfied by $\mathcal{A}^*_{\mathcal{R},\alpha}$. □

3.1.2 Normalization rules

We now introduce the first formalism that we use to define approximations: *normalization rules*. The general form for normalization rules is the following: $[s \to x] \to [l_1 \to x_1, \ldots, l_n \to x_n]$ with $s, l_1, \ldots, n_n \in \mathcal{T}(\mathcal{F} \cup \mathcal{Q}, \mathcal{X})$ and $x, x_1, \ldots, x_n \in \mathcal{X} \cup \mathcal{Q}$. The term $[s \to x]$ is a pattern to be matched over the new transitions $t \to q'$ obtained by completion and $[l_1 \to x_1, \ldots, l_n \to x_n]$ are rules used to normalize t. The syntactical constraint for those rules is the following: either $x_i \in \mathcal{Q}$ or $x_i \in \mathcal{V}ar(l_i) \cup \mathcal{V}ar(s) \cup \{x\}$. To normalize a transition of the form $t \to q'$, we match s on t and x on q', obtain a substitution σ from the matching and then we normalize t with the rewrite system $\{l_1\sigma \to r_1\sigma, \ldots, l_n\sigma \to r_n\sigma\}$ where $r_1\sigma, \ldots, r_n\sigma$ are necessarily states.

Example 68 (Application of a normalization rule) *Let us show how to normalize the transition $f(h(q_1), g(q_2)) \to q_3$ with the normalization rule $[f(x, g(y)) \to z] \to [g(u) \to z]$. We first match $f(x, g(y))$ on $f(h(q_1), g(q_2))$ and z on q_3. This gives a substitution $\sigma = \{x \mapsto h(q_1), y \mapsto q_2, z \mapsto q_3\}$. The set of rewrite rules instantiated by σ is thus $[g(u) \to q_3]$. Finally, using those instantiated rewrite rules on the left-hand side of the transition to normalize, i.e. $f(h(q_1), g(q_2))$, the subterm $g(q_2)$ can be rewritten into q_3. Hence, the transition $f(h(q_1), g(q_2)) \to q_3$ will be normalized into a normalized transition $g(q_2) \to q_3$ and a partially normalized transition $f(h(q_1), q_3) \to q_3$.*

3.1.3 The exact case

The aim of this part is to refine the previous result and show that some of the known regular classes of descendants can be computed using the tree automata completion algorithm and some particular abstraction functions. In a first part, we give some sufficient conditions on the abstraction function α so that completion is exact w.r.t. $\mathcal{R}^*(\mathcal{S})$. Then we will see how many regular classes of the literature can be expressed using abstraction functions satisfying those conditions.

During a completion step, for a rewrite rule $l \to r$, adding a transition $r\sigma \to q$ for a regular language substitution σ such that $l\sigma \to^*_\Delta q$ is not necessarily exact and may lead to an over-approximation of $\mathcal{R}^*(\mathcal{S})$. This can be the case in particular when r is not linear. This is detailed in the following example.

Example 69 *Let $\mathcal{R} = \{f(x) \to g(x,x)\}$ be a non right-linear TRS and let $\mathcal{A}$ be the tree automaton such that $\mathcal{Q}_f = \{q_0\}$ and $\Delta = \{f(q_1) \to q_0, a \to q_1, b \to q_1\}$. Note that $\mathcal{A}$ is deterministic and that $\mathcal{L}(\mathcal{A}) = \{f(a), f(b)\}$ is finite. However, the completed automaton $\mathcal{A}^1_{\mathcal{R},\alpha} = \mathcal{A}^*_{\mathcal{R},\alpha}$ (for any abstraction function α) has a new transition $g(q_1,q_1) \to q_0$ and the recognized language becomes $\mathcal{L}(\mathcal{A}^*_{\mathcal{R},\alpha}) = \{f(a), f(b), g(a,a), g(a,b), g(b,a), g(b,b)\}$, a* superset *of $\mathcal{R}^*(\mathcal{L}(\mathcal{A})) = \{f(a), f(b), g(a,a), g(b,b)\}$.*

Note that this problem trivially disappears if every state q of the automaton recognizes exactly one term. Hence, in the previous example, on an automaton $\mathcal{A}'$ s.t. the set of final states $\mathcal{Q}_f$ is $\{q_0, q'_0\}$ and the set of transitions is $\Delta = \{f(q_1) \to q_0, f(q_2) \to q'_0, a \to q_1, b \to q_2\}$, completion would produce two transitions $g(q_1,q_1) \to q_0$ and $g(q_2,q_2) \to q'_0$ recognizing exactly $\mathcal{R}^*(\mathcal{L}(\mathcal{A}'))$. Instead of requiring that every state should recognize exactly one term, we can somewhat relax this constraint by requiring that every state q should recognize exactly one term t and any of its $\mathcal{R}$-descendants. Thus, in example 69, if $\mathcal{R} = \{f(x) \to g(x,x), a \to b\}$ the language recognized by $\mathcal{A}^*_{\mathcal{R},\alpha}$ is still $\mathcal{L}(\mathcal{A}^*_{\mathcal{R},\alpha}) = \{f(a), f(b), g(a,a), g(a,b), g(b,a), g(b,b)\}$ but this time exactly covers $\mathcal{R}^*(\mathcal{L}(\mathcal{A}))$. This weaker condition will be of interest in the following since completion steps iteratively add some terms (which are $\mathcal{R}$-descendants) in the languages recognized by every states. Hence, under some assumptions, this condition is preserved by completion steps. Finally, those restrictions can trivially be left if the TRS is right-linear. The following definition formalizes all those aspects.

Definition 70 (Right-coherence condition) *A TRS $\mathcal{R}$ and a tree automaton $\mathcal{A} = \langle \mathcal{F}, \mathcal{Q}, \mathcal{Q}_f, \Delta \rangle$ satisfy the right-coherence condition if*

1. *$\mathcal{R}$ is right-linear, or*
2. *$\forall q \in \mathcal{Q} : \exists t \in \mathcal{T}(\mathcal{F}) : \mathcal{L}(\mathcal{A}, q) \subseteq \mathcal{R}^*(t)$*

$\diamond$

This condition focuses only on the *initial* automaton and not on the completed one. Note that, this condition is trivially satisfied (using the second case) by any tree automaton recognizing a finite language. This will be useful in the following theorems for defining regular classes of $\mathcal{R}^*(\mathcal{S})$ for finite sets $\mathcal{S}$.

Lemma 71 *Let $\mathcal{R}$ be a TRS, $l \to r \in \mathcal{R}$ be a rewrite rule with $r \notin \mathcal{X}$ and $\mathcal{A} = \langle \mathcal{F}, \mathcal{Q}, \mathcal{Q}_f, \Delta \rangle$ a tree automaton without dead states. Let $\sigma : \mathcal{X} \mapsto \mathcal{Q}$ be a substitution, such that $l\sigma \to^*_{\mathcal{A}} q$ with $q \in \mathcal{Q}$.*

If $\mathcal{R}$ and $\mathcal{A}$ satisfy the right-coherence condition, for all $t \in \mathcal{T}(\mathcal{F})$ s.t. $t \to^*_{\mathcal{A}} r\sigma$ then there exists $\delta : \mathcal{X} \mapsto \mathcal{T}(\mathcal{F})$ s.t. $l\delta \in \mathcal{T}(\mathcal{F})$, $l\delta \to_{\mathcal{A}}{}^* l\sigma \to_{\mathcal{A}}{}^* q$ and $l\delta \to^*_{\mathcal{R}} r\delta \to^*_{\mathcal{R}} t$.

PROOF. Let $\{p_1, \ldots, p_n\} = \mathcal{P}os_{\mathcal{X}}(r)$ and $\forall i = 1 \ldots n : x_i = r|_{p_i}$. Note that if there exist p_i and p_j s.t. $r|_{p_i} = r|_{p_j}$ then $x_i = x_j$. Let $\{y_1, \ldots, y_m\} = \mathcal{V}ar(l) \setminus \mathcal{V}ar(r)$ be the set of variables of l that do not occur in r. Note that for l it is not necessary to distinguish the multiple occurrences of non linear variables. Let $q_1, \ldots, q_n, q'_1, \ldots, q'_m \in \mathcal{Q}$ be the states such that $\sigma = \{x_1 \mapsto q_1, \ldots x_n \mapsto q_n, y_1 \mapsto q'_1, \ldots, y_m \mapsto q'_m\}$ and we obviously have $q_i = q_j$ for all $i, j \in \{1, \ldots, n\}$ such that $x_i = x_j$ since σ is a function. Thus $r = r[x_1, \ldots, x_n]$ and $r\sigma = r[q_1, \ldots, q_n]$. On the other hand, by construction of tree automata $t \rightarrow^*_{\mathcal{A}} r\sigma = r[q_1, \ldots, q_n]$ implies that there exist $t_1, \ldots, t_n \in \mathcal{T}(\mathcal{F})$ such that $t = r[t_1, \ldots, t_n]$ and $t_1 \rightarrow^*_{\mathcal{A}} q_1, \ldots, t_n \rightarrow^*_{\mathcal{A}} q_n$.

Now our aim is to show that there exists a substitution δ such that $l\delta \rightarrow r\delta = t$. The substitution δ we build is such that $\delta = \delta_r \cup \delta_l$ where $\mathcal{D}om(\delta_r) = \{x_1, \ldots, x_m\}$ and $\mathcal{D}om(\delta_l) = \{y_1, \ldots, y_m\}$. First we show how to construct δ_r. Let δ_r be the relation $\{x_1 \mapsto t_1, \ldots, x_n \mapsto t_n\}$. Note that δ_r is a substitution (i.e. a function) if and only if there is no $i, j \in \{1, \ldots, n\}$ such that $x_i = x_j$ and $t_i \neq t_j$. This is of course trivially the case if r is linear. Otherwise, we may have $x_i = x_j$, $t_i \rightarrow^*_{\mathcal{A}} q_i$, $t_j \rightarrow^*_{\mathcal{A}} q_j$, $q_i = q_j$ but $t_i \neq t_j$ and thus δ_r would not be a function. However, if condition 2. of definition 70 is satisfied then we know that $\forall i = 1 \ldots n : \exists t'_i : \mathcal{L}(\mathcal{A}, q_i) \subseteq \mathcal{R}^*(\{t'_i\})$. Hence, every term which is recognized into q_i is either t'_i or one of its descendants. As a result, since $q_i = q_j$ we have that $t_i, t_j \in \mathcal{R}^*(t'_i)$. In this case, if we replace t_i and t_j by t'_i in δ_r (and proceed similarly for every other occurrence of a non linear variable), we obtain a valid substitution δ_r such that $r\delta_r \rightarrow_{\mathcal{R}}{}^* r[t_1, \ldots, t_n]$. Thus, using case 1. or case 2. of definition 70 leads to the same property: we have built a substitution δ_r such that $r\delta_r \rightarrow_{\mathcal{R}}{}^* r[t_1, \ldots, t_n]$ and $r[t_1, \ldots, t_n] \rightarrow^*_{\mathcal{A}} r[q_1, \ldots, q_n] = r\sigma$.

Now, recall that $\delta = \delta_r \cup \delta_l$. For δ_l we construct a substitution mapping the variables of l not occurring in r to any term recognized by the corresponding state in σ, i.e. $\delta_l = \bigcup_{i=1\ldots m}\{y_i \mapsto u'_i \mid \exists u'_i \in \mathcal{T}(\mathcal{F})$ s.t. $u'_i \rightarrow^*_{\mathcal{A}} q'_i$ and $q'_i = \sigma(y_i)\}$. Note that the existence of u'_i s.t. $u'_i \rightarrow^*_{\mathcal{A}} q'_i$ is guaranteed by the fact that q'_i is a state of $\mathcal{A}$ and there is no dead state in $\mathcal{A}$. The relation δ_l is a functional substitution and so is δ_r. Furthermore, since $\mathcal{D}om(\delta_r) \cap \mathcal{D}om(\delta_l) = \emptyset$ then δ is a substitution. Finally. we have $l\delta \in \mathcal{T}(\mathcal{F})$, $l\delta \rightarrow^*_{\mathcal{A}} l\sigma$ and $l\delta \rightarrow^*_{\mathcal{R}} r\delta \rightarrow^*_{\mathcal{R}} r[t_1, \ldots, t_n] = t$. □

We now introduce *coherent abstraction function* which define some subclasses of completion algorithms for which the automaton completion algorithm is exact. Informally, an abstraction function is coherent with regards to a tree automaton $\mathcal{A}$ and a term rewriting system $\mathcal{R}$ if for every configuration t and every state q such that α maps t to q, either q is not a state of $\mathcal{A}$ (it is a new state) or terms recognized by q in $\mathcal{A}$ are either a term t' recognized by t (i.e. $t' \rightarrow_{\mathcal{A}}{}^* t$) or $\mathcal{R}$-descendants of t'.

Definition 72 (Coherent abstraction function) *Let $\mathcal{R}$ be a TRS, $\mathcal{A} = \langle \mathcal{F}, \mathcal{Q}, \mathcal{Q}_f, \Delta \rangle$ be a tree automaton and α be an abstraction function. The function α is said to be* coherent *with $\mathcal{R}$ and $\mathcal{A}$ if for all $t \in \mathcal{D}om(\alpha)$, for all $q \in \mathcal{Q} \cap \mathcal{R}an(\alpha)$ if $\alpha(t) = q$ then $t \rightarrow q \in \mathcal{A}$ and there exists a term $t' \in \mathcal{T}(\mathcal{F})$ called the* representative *of q s.t. $t' \rightarrow_{\mathcal{A}}{}^* t$ and $\mathcal{L}(\mathcal{A}, q) \subseteq \mathcal{R}^*(\{t'\})$.* ◇

Note that any abstraction function α mapping any term to a state not in $\mathcal{Q}$ is trivially coherent. The following lemma shows some additional properties on the coherent abstraction

functions.

Lemma 73 *Let α be an abstraction function coherent with a TRS $\mathcal{R}$ and a tree automaton $\mathcal{A}$. For all $t \in \mathcal{T}(\mathcal{F} \cup \mathcal{Q})$ and $q \in \mathcal{A}$ s.t. $top_\alpha(t) = q$ then we have $t \rightarrow^*_{\mathcal{A}} q$ and the representative t' of q is such that $t' \rightarrow^*_{\mathcal{A}} t$.*

PROOF. We proceed by induction on the height of t:

- if $t = a$ a constant. From $top_\alpha(a) = q$ we get that $\alpha(a) = q$ and from definition 72, we obtain that $a \rightarrow q \in \mathcal{A}$. Furthermore, since in that case the representative t' of q is a, we trivially have $t' = a \rightarrow^*_{\mathcal{A}} a$.
- if $t = f(t_1, \ldots t_n)$ then $top_\alpha(f(t_1, \ldots, t_n)) = \alpha(f(top_\alpha(t_1), \ldots, top_\alpha(t_n))) = q$. Hence, there exist states $q_1, \ldots, q_n$ such that $\forall i = 1 \ldots n : top_\alpha(t_i) = q_i$ and $\alpha(f(q_1, \ldots, q_n)) = q$. By definition 72, from $\alpha(f(q_1, \ldots, q_n)) = q$, we get that $f(q_1, \ldots, q_n) \rightarrow q \in \mathcal{A}$ and thus $q_1, \ldots, q_n \in \mathcal{A}$. Applying the induction hypothesis on $q_1, \ldots, q_n$ and $\forall i = 1 \ldots n : top_\alpha(t_i) = q_i$ we get that $\forall i = 1 \ldots n : t_i \rightarrow^*_{\mathcal{A}} q_i$ and the corresponding representatives t'_i are such that $t'_i \rightarrow^*_{\mathcal{A}} t_i$. From $t_i \rightarrow^*_{\mathcal{A}} q_i$ and from $f(q_1, \ldots, q_n) \rightarrow q \in \mathcal{A}$ we get that $t = f(t_1, \ldots, t_n) \rightarrow^*_{\mathcal{A}} q$. From $t'_i \rightarrow^*_{\mathcal{A}} t_i$ we get that $t' = f(t'_1, \ldots, t'_n) \rightarrow^*_{\mathcal{A}} f(t_1, \ldots, t_n) = t$.

□

The next lemma gives some sufficient conditions on a completion step to ensure that terms recognized by $\mathcal{C}_{\mathcal{R},\alpha}(\mathcal{A})$ are all reachable terms. One of the condition is that α has to be *injective* i.e. there is no couple of distinct terms $t, t' \in \mathcal{D}om(\alpha)$ such that $\alpha(t) = \alpha(t')$. The fact that the conditions used in this lemma are all necessary will be shown on examples in the following.

Lemma 74 *Let $\mathcal{R}$ be a TRS, $\mathcal{A} = \langle \mathcal{F}, \mathcal{Q}, \mathcal{Q}_f, \Delta \rangle$ be a tree automaton and α be an abstraction function. If $\mathcal{R}$ and $\mathcal{A}$ satisfy the right-coherence condition and if α is injective and coherent with regards to $\mathcal{R}$ and $\mathcal{A}$ then:*

$$\forall t \in \mathcal{T}(\mathcal{F}), \forall q \in \mathcal{Q} : t \in \mathcal{L}(\mathcal{C}_{\mathcal{R},\alpha}(\mathcal{A}), q) \implies t \in \mathcal{R}^*(\mathcal{L}(A, q))$$

PROOF. For terms t such that $t \in \mathcal{L}(\mathcal{A}, q)$, we trivially have that $t \in \mathcal{R}^*(\mathcal{L}(\mathcal{A}, q))$. So, we can restrict the proof to terms t such that $t \notin \mathcal{L}(\mathcal{A}, q)$. Similarly, we can distinguish two other particular cases where $t \rightarrow^*_\Delta q$ is in fact of the form $t \rightarrow^*_\Delta q' \rightarrow q$.

- either $q' \rightarrow q \in \mathcal{A}$ and $t \notin \mathcal{L}(\mathcal{A}, q)$. Since $q' \rightarrow q$ is in $\mathcal{A}$, we have $\mathcal{L}(\mathcal{A}, q') \subseteq \mathcal{L}(\mathcal{A}, q)$ and $\mathcal{R}^*(\mathcal{L}(\mathcal{A}, q')) \subseteq \mathcal{R}^*(\mathcal{L}(\mathcal{A}, q)$. Hence, to prove that $t \in \mathcal{R}^*(\mathcal{L}(\mathcal{A}, q)$ it is enough to prove that $t \in \mathcal{R}^*(\mathcal{L}(\mathcal{A}, q')$ from $t \rightarrow^*_\Delta q'$. We can proceed similarly to remove every epsilon transition of the form $q_1 \rightarrow q_2$ which are already in $\mathcal{A}$ and prove that $t \rightarrow^*_\Delta q'$ implies that $t \in \mathcal{R}^*(\mathcal{L}(\mathcal{A}, q')$ in the general case.
- either $t \in \mathcal{L}(\mathcal{A}, q')$, $t \notin \mathcal{L}(\mathcal{A}, q)$ and $q' \rightarrow q \in \mathcal{C}_{\mathcal{R},\alpha}(\mathcal{A})$. In that case, the completion step producing $\mathcal{C}_{\mathcal{R},\alpha}(\mathcal{A})$ from $\mathcal{A}$ necessarily builds a critical pair of the form $l\sigma \rightarrow_{\mathcal{R}} r\sigma = q'$ and $l\sigma \rightarrow_{\mathcal{A}}{}^* q$ where $l \rightarrow r \in \mathcal{R}$. In that case, we necessarily have

$l = C[x]$ and $r = x$ where $x \in \mathcal{V}ar(l)$ and $\sigma = \{x \mapsto q\} \cup \sigma'$. Hence, we have $l\sigma = C[q']\sigma' \rightarrow_{\mathcal{A}}{}^* q$ and since $t \in \mathcal{L}(\mathcal{A}, q')$, we have $C[t] \rightarrow_{\mathcal{A}}{}^* C[q'] \rightarrow_{\mathcal{A}}{}^* q$ and thus $C[t] \in \mathcal{L}(\mathcal{A}, q)$. Finally since the rule $l \rightarrow r$ is of the form $C[x] \rightarrow x$ we get that $t \in \mathcal{R}^*(\mathcal{L}(\mathcal{A}, q))$.

Now for the general case, we proceed by induction over the height of t.

- If height of t is 0 then $t = a$ where a is a constant. Since $a \in \mathcal{L}(\mathcal{C}_{\mathcal{R},\alpha}(\mathcal{A}), q)$ we have $a \rightarrow q \in \mathcal{C}_{\mathcal{R},\alpha}(\mathcal{A})$. Since $a \notin \mathcal{L}(\mathcal{A}, q)$ then the completion step producing $\mathcal{C}_{\mathcal{R},\alpha}(\mathcal{A})$ from $\mathcal{A}$ necessarily builds a critical pair of the form $l\sigma \rightarrow_{\mathcal{R}} a$ and $l\sigma \rightarrow_{\mathcal{A}}{}^* q$ where $l \rightarrow a \in \mathcal{R}$. By lemma 71, we obtain that there exists a substitution δ such that $l\delta \in \mathcal{T}(\mathcal{F})$, $l\delta \rightarrow_{\mathcal{A}}{}^* q$ and $l\delta \rightarrow_{\mathcal{R}}{}^* a$, hence $a \in \mathcal{R}^*(\mathcal{L}(\mathcal{A}, q))$.

- Now, we assume that the property is true for terms of height n. Let us prove that the property also holds for terms of height $n + 1$. Let t be a term of height $n + 1$ such that $t \in \mathcal{L}(\mathcal{C}_{\mathcal{R},\alpha}(\mathcal{A}), q)$. Let $f \in \mathcal{F}$ and $t_1, \ldots, t_n \in \mathcal{T}(\mathcal{F})$ terms of height lesser or equal to n such that $t = f(t_1, \ldots, t_n)$. By construction of tree automata, $t \in \mathcal{L}(\mathcal{C}_{\mathcal{R},\alpha}(\mathcal{A}), q)$ implies that there exist states $q_1, \ldots, q_n$ such that $f(t_1, \ldots, t_n) \rightarrow^*_{\mathcal{C}_{\mathcal{R},\alpha}(\mathcal{A})} f(q_1, \ldots, q_n) \rightarrow^*_{\mathcal{C}_{\mathcal{R},\alpha}(\mathcal{A})} q$. Then by cases on $f(q_1, \ldots, q_n) \rightarrow q \in \mathcal{A}$, we obtain:

  - Assume that $f(q_1, \ldots, q_n) \rightarrow q \in \mathcal{A}$. Then $q_1, \ldots, q_n \in \mathcal{A}$ and by induction hypothesis we get that $t_i \in \mathcal{R}^*(\mathcal{L}(\mathcal{A}, q_i))$ for $i = 1 \ldots n$. Hence, there exist terms t'_i such that $t'_i \rightarrow_{\mathcal{A}}{}^* q_i$ and $t'_i \rightarrow_{\mathcal{R}} t_i$. Hence $f(t'_1, \ldots t'_n) \rightarrow_{\mathcal{A}}{}^* f(q_1, \ldots, q_n) \rightarrow_{\mathcal{A}} q$ and $f(t'_1, \ldots t'_n) \rightarrow_{\mathcal{R}}{}^* f(t_1, \ldots, t_n)$, i.e. $f(t'_1, \ldots, t'_n) \in \mathcal{L}(\mathcal{A}, q)$ and $t \in \mathcal{R}^*(\mathcal{L}(\mathcal{A}, q))$.
  - Now, assume that $f(q_1, \ldots, q_n) \rightarrow q \notin \mathcal{A}$. Thus, transition $f(q_1, \ldots, q_n) \rightarrow q$ has been added to $\mathcal{A}$ by the completion step. Hence there exists terms $t''_1, \ldots, t''_n$ such that $\forall i = 1 \ldots n : top_\alpha(t''_i) = q_i$ and there is either a critical pair of the form **(a)** $l\sigma \rightarrow_{\mathcal{R}} C[f(t''_1, \ldots, t''_n)]$, with $top_\alpha(f(t''_1, \ldots, t''_n)) = q$ and $l\sigma \rightarrow_{\mathcal{A}}{}^* q'$ or **(b)** $l\sigma \rightarrow_{\mathcal{R}} f(t''_1, \ldots, t''_n)$ and $l\sigma \rightarrow_{\mathcal{A}}{}^* q$. Let us continue the proof on those two cases:

    (a) Assume that there is a critical pair of the form $l\sigma \rightarrow_{\mathcal{R}} C[f(t''_1, \ldots, t''_n)]$ with $top_\alpha(f(t''_1, \ldots, t''_n)) = q$. From the fact that $\{f(q_1, \ldots, q_n) \rightarrow q\} \subseteq Norm_\alpha(f(t''_1, \ldots, t''_n) \rightarrow q)$, we get that $\alpha(f(q_1, \ldots, q_n)) = q$. But, $q \in \mathcal{A}$ and since α is coherent with $\mathcal{R}$ and $\mathcal{A}$ we get that $f(q_1, \ldots, q_n) \rightarrow q \in \mathcal{A}$, which contradicts the hypothesis that $f(q_1, \ldots, q_n) \rightarrow q \notin \mathcal{A}$.

    (b) Assume that there is a critical pair of the form $l\sigma \rightarrow_{\mathcal{R}} r\sigma = f(t''_1, \ldots, t''_n)$ and $l\sigma \rightarrow_{\mathcal{A}}{}^* q$. First, let us prove that there exist terms $s_1, \ldots, s_n$ s.t. $\forall i = 1 \ldots n : s_i \rightarrow_{\mathcal{R}}{}^* t_i$ and $s_i \rightarrow^*_{\mathcal{C}_{\mathcal{R},\alpha}(\mathcal{A})} t''_i$. Doing so permits to prove that there exists a term $f(s_1, \ldots, s_n)$ such that $f(s_1, \ldots, s_n) \rightarrow_{\mathcal{R}}{}^* f(t_1, \ldots, t_n) = t$ and $f(s_1, \ldots, s_n) \rightarrow^*_{\mathcal{C}_{\mathcal{R},\alpha}(\mathcal{A})} f(t''_1, \ldots, t''_n) = r\sigma$. By cases on $q_i \in \mathcal{A}$, we obtain:

* Assume that q_i occurs in $\mathcal{A}$. Since $top_\alpha(t''_i) = q_i$ and α is injective, we know that there exists a unique term u_i such that $\alpha(u_i) = q_i$. Furthermore, since $q_i \in \mathcal{A}$ and α coherent with $\mathcal{R}$ and $\mathcal{A}$ we get from definition 72 that $\exists s_i \in \mathcal{T}(\mathcal{F})$ s.t. $s_i \rightarrow^*_{\mathcal{A}} u_i$ and $\mathcal{L}(\mathcal{A}, q_i) \subseteq \mathcal{R}^*(\{s_i\})$ where s_i is the representative of q_i. On the other hand, by applying the induction hypothesis to $t_i \rightarrow^*_{\mathcal{C}_{\mathcal{R},\alpha}(\mathcal{A})} q_i$, we get that $t_i \in \mathcal{R}^*(\mathcal{L}(\mathcal{A}, q_i))$. Combining the two, we get that $t_i \in \mathcal{R}^*(\mathcal{R}^*(\{s_i\}))$ i.e. $t_i \in \mathcal{R}^*(\{s_i\})$. Thus $s_i \rightarrow_{\mathcal{R}}{}^* t_i$. Now, what remains to be proved is that $\forall i = 1 \ldots n : s_i \rightarrow^*_{\mathcal{A}} t''_i$. Applying lemma 73 on $q_i \in \mathcal{A}$ where $top_\alpha(t''_i) = q_i$, α is coherent with $\mathcal{R}$ and $\mathcal{A}$, and s_i is the representative of q_i, we get that $s_i \rightarrow^*_{\mathcal{A}} t''_i \rightarrow^*_{\mathcal{A}} q_i$. Hence, we have $s_i \rightarrow^*_{\mathcal{C}_{\mathcal{R},\alpha}(\mathcal{A})} t''_i \rightarrow^*_{\mathcal{C}_{\mathcal{R},\alpha}(\mathcal{A})} q_i$ and $s_i \rightarrow_{\mathcal{R}}{}^* t_i$.
* if q_i does not occur in $\mathcal{A}$ then the only rewriting from t_i to q_i is necessarily $t_i \rightarrow^*_{\mathcal{C}_{\mathcal{R},\alpha}(\mathcal{A})} t''_i \rightarrow^*_{\mathcal{C}_{\mathcal{R},\alpha}(\mathcal{A})} q_i$, since the only rewriting path to q_i, which is $t''_i \rightarrow^*_{\mathcal{C}_{\mathcal{R},\alpha}(\mathcal{A})} q_i$, has been added by the current completion step (α is injective). Hence $t_i \rightarrow^*_{\mathcal{C}_{\mathcal{R},\alpha}(\mathcal{A})} t''_i$. In that case let $s_i = t_i$.

Thus, we have $f(s_1, \ldots, s_n) \rightarrow^*_{\mathcal{C}_{\mathcal{R},\alpha}(\mathcal{A})} f(t''_1, \ldots, t''_n) = r\sigma, r\sigma \rightarrow^*_{\mathcal{C}_{\mathcal{R},\alpha}(\mathcal{A})} q$ and $f(s_1, \ldots, s_n) \rightarrow_{\mathcal{R}}{}^* f(t_1, \ldots, t_n) = t$. By, applying lemma 71 to term $f(s_1, \ldots, s_n) \rightarrow^*_{\mathcal{C}_{\mathcal{R},\alpha}(\mathcal{A})} r\sigma$, we get that there exists δ such that $l\delta \in \mathcal{T}(\mathcal{F})$, $l\delta \rightarrow^*_{\mathcal{C}_{\mathcal{R},\alpha}(\mathcal{A})} l\sigma \rightarrow^*_{\mathcal{C}_{\mathcal{R},\alpha}(\mathcal{A})} q$ and $l\delta \rightarrow_{\mathcal{R}}{}^* r\delta \rightarrow_{\mathcal{R}}{}^* f(s_1, \ldots, s_n)$. Let $\{x_1, \ldots, x_m\}$ be the domain of σ. For all $i = 1 \ldots m$ let $q'_i = x_i\sigma$ and $u_i = \delta x_i$. Since $u_i \in \mathcal{L}(\mathcal{C}_{\mathcal{R},\alpha}(\mathcal{A}), q'_i)$ and u_i is a subterm of t, we can apply the induction hypothesis on u_i and we obtain that $u_i \in \mathcal{R}^*(\mathcal{L}(\mathcal{A}, q'_i))$. Let v_i be a term of $\mathcal{L}(\mathcal{A}, q'_i)$ such that $v_i \rightarrow_{\mathcal{R}}{}^* u_i$. Finally let $\delta' = \{x_1 \mapsto v_1, \ldots, x_m \mapsto v_m\}$. We have $l\delta' \rightarrow_{\mathcal{A}}{}^* q$, $l\delta' \rightarrow_{\mathcal{R}}{}^* l\delta \rightarrow_{\mathcal{R}}{}^* r\delta \rightarrow_{\mathcal{R}}{}^* t$.

□

The following theorem states that under some conditions, iterating completion steps builds an automaton $\mathcal{A}^n_{\mathcal{R},\alpha}$ recognizing only reachable terms. By extension, this is of course also true for $\mathcal{A}^*_{\mathcal{R},\alpha}$ when it can be built. However, this theorem is given using $\mathcal{A}^n_{\mathcal{R},\alpha}$ for all n because it is more general. Furthermore it can be used to under-approximate $\mathcal{R}^*(\mathcal{L}(\mathcal{A}))$ by computing $\mathcal{A}^n_{\mathcal{R},\alpha}$ for a fixed n when completion does not terminate ($\mathcal{A}^*_{\mathcal{R},\alpha}$ does not exist).

Theorem 75 *Let $\mathcal{R}$ be a TRS, $\mathcal{A} = \langle \mathcal{F}, \mathcal{Q}, \mathcal{Q}_f, \Delta \rangle$ a tree automaton and α an injective abstraction function coherent with $\mathcal{R}$ and $\mathcal{A}$. If $\mathcal{R}$ and $\mathcal{A}$ satisfy the right-coherence condition then*

$$\forall n \in \mathbb{N} : \mathcal{L}(\mathcal{A}^n_{\mathcal{R},\alpha}) \subseteq \mathcal{R}^*(\mathcal{L}(\mathcal{A}))$$

PROOF. We proceed by induction on the number of completion steps: n. If $n = 0$ we have $\mathcal{A}^0_{\mathcal{R},\alpha} = \mathcal{A}$ and thus $\mathcal{L}(\mathcal{A}) \subseteq \mathcal{R}^*(\mathcal{L}(\mathcal{A}))$. Then, we assume that the property holds for n completion steps and we prove that it holds for $n + 1$. Let us denote by $\mathcal{B}$ the tree

automaton $\mathcal{A}^1_{\mathcal{R},\alpha} = \mathcal{C}_{\mathcal{R},\alpha}(\mathcal{A})$. Then, the proof is done by using the induction hypothesis on $\mathcal{B}$ since we have $\mathcal{A}^{n+1}_{\mathcal{R},\alpha} = \mathcal{B}^n_{\mathcal{R},\alpha}$. By lemma 74, we know that for every state $q \in \mathcal{Q}$, and for every term $t \in \mathcal{L}(\mathcal{B}, q) = \mathcal{L}(\mathcal{C}_{\mathcal{R},\alpha}(\mathcal{A}), q)$ we have $t \in \mathcal{R}^*(\mathcal{L}(\mathcal{A}, q))$. This property is true in particular for final states, thus we have: $\mathcal{L}(\mathcal{B}) \subseteq \mathcal{R}^*(\mathcal{L}(\mathcal{A}))$. Now, in order to use the induction hypothesis on $\mathcal{B}$, we need to prove that $\mathcal{B}$ fulfills the conditions of the theorem, i.e. that **(a)** α is coherent with $\mathcal{R}$ and $\mathcal{B}$ and that **(b)** $\mathcal{R}$ and $\mathcal{B}$ satisfy the right-coherence condition.

(a) For every left-hand side of a normalized transition $t \in \mathcal{T}(\mathcal{F} \cup \mathcal{Q})$ such that $\alpha(t) = q$,

- if $q \in \mathcal{A}$ then we had already $t \to q \in \mathcal{A}$ so $t \to q \in \mathcal{B}$. Since α is coherent with $\mathcal{R}$ and $\mathcal{A}$, we know that $\mathcal{L}(\mathcal{A}, q) \subseteq \mathcal{R}^*(\{t' \mid t' \to_{\mathcal{A}}{}^* t\})$. By applying the $\mathcal{R}^*$ operator to both sides of the previous inequality, we obtain that $\mathcal{R}^*(\mathcal{L}(\mathcal{A}, q)) \subseteq \mathcal{R}^*(\mathcal{R}^*(\{t' \mid t' \to_{\mathcal{A}}{}^* t\}))$. On the other hand, by lemma 74, we get that $\mathcal{L}(\mathcal{B}, q) \subseteq \mathcal{R}^*(\mathcal{L}(\mathcal{A}, q))$. Note that $\mathcal{R}^*(\mathcal{R}^*(\mathcal{S})) = \mathcal{R}^*(\mathcal{S})$ for any set $\mathcal{S}$, hence we have: $\mathcal{L}(\mathcal{B}, q) \subseteq \mathcal{R}^*(\{t' \mid t' \to_{\mathcal{A}}{}^* t\})$. Moreover, since $\{t' \mid t' \to_{\mathcal{A}}{}^* t\} \subseteq \{t' \mid t' \to^*_{\mathcal{B}} t\}$ we have $\mathcal{R}^*(\{t' \mid t' \to_{\mathcal{A}}{}^* t\}) \subseteq \mathcal{R}^*(\{t' \mid t' \to^*_{\mathcal{B}} t\})$ and by transitivity of $\subseteq$, we get that $\mathcal{L}(\mathcal{B}, q) \subseteq \mathcal{R}^*(\{t' \mid t' \to^*_{\mathcal{B}} t\})$.
- if $q \notin \mathcal{A}$ but $q \in \mathcal{B}$ then q is a state that has been introduced by $\mathcal{C}_{\mathcal{R},\alpha}(\mathcal{A})$ and since α is injective we know that t is the unique left-hand side of a normalized transition s.t. $t \to^*_{\mathcal{B}} q$. Hence, $\mathcal{L}(\mathcal{B}, q) = \{t' \mid t' \to^*_{\mathcal{B}} t\} \subseteq \mathcal{R}^*(\{t' \mid t' \to^*_{\mathcal{B}} t\})$.

(b) We know by hypothesis that $\mathcal{R}$ and $\mathcal{A}$ satisfy the right-coherence condition. If $\mathcal{R}$ and $\mathcal{A}$ satisfy the condition because $\mathcal{R}$ is right-linear then it will clearly be the case for $\mathcal{R}$ and $\mathcal{B}$. Otherwise, by hypothesis, we know that every term t' recognized by q in $\mathcal{A}$ has a common ancestor t such that $t \to_{\mathcal{R}}{}^* t'$, i.e. $\forall q \in \mathcal{A} : \exists t \in \mathcal{T}(\mathcal{F}) : \mathcal{L}(\mathcal{A}, q) \subseteq \mathcal{R}^*(t)$. Now, we have to prove that it is also the case for $\mathcal{B}$.

- if $q \in \mathcal{A}$ then we know that $\exists t \in \mathcal{T}(\mathcal{F}) : \mathcal{L}(\mathcal{A}, q) \subseteq \mathcal{R}^*(t)$. From lemma 74 we get that every term recognized by q in $\mathcal{B}$ has an ancestor in the terms recognized by q in $\mathcal{A}$, i.e. $\forall t_{\mathcal{B}} \in \mathcal{L}(\mathcal{B}, q) : \exists t_{\mathcal{A}} \in \mathcal{L}(\mathcal{A}, q)$ s.t. $t_{\mathcal{A}} \to_{\mathcal{R}}{}^* t_{\mathcal{B}}$. Since, by hypothesis on $\mathcal{A}$, all the terms recognized by have a common ancestor (w.r.t $\mathcal{R}$) t , it is also the case for terms recognized by q in $\mathcal{B}$ (and it is the same ancestor t), i.e. $t \to_{\mathcal{R}}{}^* t_{\mathcal{A}} \to_{\mathcal{R}}{}^* t_{\mathcal{B}}$. Hence, $\forall q \in \mathcal{B} : \exists t \in \mathcal{T}(\mathcal{F}) : \mathcal{L}(\mathcal{B}, q) \subseteq \mathcal{R}^*(t)$.
- if $q \notin \mathcal{A}$ but $q \in \mathcal{B}$ then q is a state that has been introduced by $\mathcal{C}_{\mathcal{R},\alpha}(\mathcal{A})$. Let $s \to q'$ be the new transition whose normalization has led to construction of state q, i.e. $s = C[u]$ and $top_\alpha(u) = q$. By induction on the height of u we show that $\exists t \in \mathcal{T}(\mathcal{F}) : \mathcal{L}(\mathcal{B}, q) \subseteq \mathcal{R}^*(t)$:
  - if u is a constant, since $q \notin \mathcal{A}$ and α is injective we know that $u \to q$ is the unique transition with q on the right-hand side, hence $\mathcal{L}(\mathcal{B}, q) = \{u\} \subseteq \mathcal{R}^*(u)$.
  - if $u = f(t_1, \ldots, t_n)$ then since α is injective we know that there is a unique left-hand side of a normalized transition $f(q_1, \ldots, q_n)$ such that

$f(q_1, \ldots, q_n) \rightarrow_{\mathcal{B}} q$ and $t_i \rightarrow^*_{\mathcal{B}} q_i$ for $i = 1 \ldots n$. For every state q_i, $i = 1 \ldots n$, it is possible to find a unique term t'_i such that $\mathcal{L}(\mathcal{B}, q_i) \subseteq \mathcal{R}^*(t'_i)$. If $q_i \notin \mathcal{A}$ then we use the induction hypothesis on transition $t_i \rightarrow q_i$. Otherwise, if $q_i \in \mathcal{A}$, then the proof is similar to the first case of the proof: by hypothesis we know that $\mathcal{L}(\mathcal{A}, q_i) \subseteq \mathcal{R}^*(t'_i)$ and from Lemma 74, we can lift this property to $\mathcal{B}$, i.e. $\mathcal{L}(\mathcal{B}, q_i) \subseteq \mathcal{R}^*(t'_i)$. Finally $\mathcal{L}(\mathcal{B}, q) \subseteq \mathcal{R}^*(f(t'_1, \ldots, t'_n))$

Finally, applying the induction hypothesis to $\mathcal{B}$, we get that $\mathcal{L}(\mathcal{A}^{n+1}_{\mathcal{R},\alpha}) = \mathcal{L}(\mathcal{B}^n_{\mathcal{R},\alpha}) \subseteq \mathcal{R}^*(\mathcal{L}(\mathcal{B})) \subseteq \mathcal{R}^*(\mathcal{R}^*(\mathcal{L}(\mathcal{A}))) = \mathcal{R}^*(\mathcal{L}(\mathcal{A}))$. □

Theorem 76 *Let $\mathcal{R}$ be a TRS, $\mathcal{A}$ be a tree automaton, α be an injective abstraction function coherent with $\mathcal{R}$ and $\mathcal{A}$.*

$$\mathcal{L}(\mathcal{A}^*_{\mathcal{R},\alpha}) = \mathcal{R}^*(\mathcal{L}(\mathcal{A}))$$

if $\mathcal{A}^*_{\mathcal{R},\alpha}$ exists, $\mathcal{R}$ and $\mathcal{A}$ fulfills the right-coherence condition and if $\mathcal{R}$ and $\mathcal{A}^*_{\mathcal{R},\alpha}$ fulfills the left-coherence condition.

PROOF. Direct consequence of theorems 75 and 67. □

This theorem states the general properties of $\mathcal{A}^*_{\mathcal{R},\alpha}$ but it says nothing about the existence of $\mathcal{A}^*_{\mathcal{R},\alpha}$, i.e. of termination of the completion. In the following, we give some interesting instances of this theorem as corollaries and some conditions for completion to terminate. The two first corollaries permits one to use automata completion as a rewriting tool: for any given finite initial language, tree automata completion produces exactly reachable terms. We will show in section 4.6.3 that using tree automata completion in this setting provide an efficient alternative to breadth-first search for a particular descendant.

Corollary 77 *Let $\mathcal{R}$ be a TRS, $\mathcal{A} = \langle \mathcal{F}, \mathcal{Q}, \mathcal{Q}_f, \Delta \rangle$ be a tree automaton such that $\forall q \in \mathcal{Q} : \mathcal{C}ard(\mathcal{L}(\mathcal{A}, q)) = 1$, α an injective abstraction such that $\mathcal{R}an(\alpha) \cap \mathcal{Q} = \emptyset$.*

$$\mathcal{L}(\mathcal{A}^*_{\mathcal{R},\alpha}) = \mathcal{R}^*(\mathcal{L}(\mathcal{A}))$$

if $\mathcal{A}^*_{\mathcal{R},\alpha}$ and $\mathcal{A}^*_{\mathcal{R},\alpha}$ and $\mathcal{R}$ satisfy the left-coherence condition.

PROOF. Consequence of Theorem 76. Since, $\mathcal{A}$ satisfies $\forall q \in \mathcal{Q} : \mathcal{C}ard(\mathcal{L}(\mathcal{A}, q)) = 1$, the right-coherence condition is trivially fulfilled. Similarly, since $\mathcal{R}an(\alpha) \cap \mathcal{Q} = \emptyset$, α is trivially coherent with $\mathcal{R}$ and $\mathcal{A}$. □

A direct consequence of this corollary is that applying completion to a tree automaton recognizing one term models exactly rewriting if α is injective and $\mathcal{R}$ is left-linear. Now, let us show that if any of the above restrictions is not satisfied then the completed automaton no longer recognizes exactly the set of reachable terms.

Example 78 (Left-coherence condition is necessary) *Let $\mathcal{R} = \{f(x, x) \rightarrow g(x),\ a \rightarrow b\}$, let $\mathcal{A} = \langle \mathcal{F}, \mathcal{Q}, \mathcal{Q}_f, \Delta \rangle$ be the tree automaton with $\mathcal{Q} = \{q_0, q_1, q_2\}$, $\mathcal{Q}_f = \{q_0\}$ and a set of transitions $\Delta = \{f(q_1, q_2) \rightarrow q_0,\ a \rightarrow q_1,\ b \rightarrow q_2\}$. Note that $\mathcal{A}$ is deterministic, it recognizes a finite language $\mathcal{L}(\mathcal{A}) = \{f(a, b)\}$ and it satisfies $\forall q \in \mathcal{Q} : \mathcal{C}ard(\mathcal{L}(\mathcal{A}, q)) = 1$. However, for any abstraction function α, the tree automata completion produces a unique new transition: $b \rightarrow q_1$ and the completed automaton does not recognize term $g(b)$ which is a descendant of $f(a, b)$.*

Example 79 (Unicity of initial recognized language is necessary if $\mathcal{R}$ is not right linear) *Let $\mathcal{R} = \{f(x) \to g(x,x)\}$, let $\mathcal{A} = \langle \mathcal{F}, \mathcal{Q}, \mathcal{Q}_f, \Delta \rangle$ be the tree automaton with $\mathcal{Q} = \{q_0, q_1\}$, $\mathcal{Q}_f = \{q_0\}$ and set of transitions $\Delta = \{f(q_1) \to q_0,\ a \to q_1,\ b \to q_1\}$. Note that $\mathcal{A}$ is deterministic and recognizes a finite language $\mathcal{L}(\mathcal{A}) = \{f(a), f(b)\}$. However, completion with any α produces a new transition $g(q_1, q_1) \to q_0$ and thus the completed automaton recognizes terms $g(a,b)$ and $g(b,a)$ which are not valid descendants of $f(a)$ nor of $f(b)$.*

Example 80 (Abstraction function needs to be injective) *Let $\mathcal{R} = \{f(c) \to g(a,b)\}$, let $\mathcal{A} = \langle \mathcal{F}, \mathcal{Q}, \mathcal{Q}_f, \Delta \rangle$ be the tree automaton with $\mathcal{Q} = \{q_0, q_1\}$, $\mathcal{Q}_f = \{q_0\}$ and set of transitions $\Delta = \{f(q_1) \to q_0,\ c \to q_1\}$. Note that $\mathcal{A}$ is deterministic, it recognizes a finite language $\mathcal{L}(\mathcal{A}) = \{f(c)\}$ and it satisfies $\forall q \in \mathcal{Q} : \mathcal{C}ard(\mathcal{L}(\mathcal{A}, q)) = 1$. However, tree automata completion produces a new transition $g(a,b) \to q_0$ which has to be normalized. Now assume that α is a non injective function mapping a and b to the same state q_2, i.e. $\alpha = \{a \mapsto q_2,\ b \mapsto q_2\}$, then $Norm_\alpha(g(a,b) \to q_0) = \{a \to q_2,\ b \to q_2,\ g(q_2, q_2) \to q_0\}$. Thus, the completed automaton recognizes terms $g(a,b)$ which is correct but also $g(a,a)$, $g(b,b)$ and $g(b,a)$ which are not valid descendants of $f(a)$.*

As we will see in Sections 3.1.2 and 3.2 with non regular sets of descendants, using non injective abstraction functions is a very convenient way to force completion to terminate and build over-approximations.

The following corollary will be used to give alternative proofs of results for ground TRSs (Dauchet and Tison, 1990; Brainerd, 1969), linear and semi-monadic (Coquidé et al., 1991), linear and "decreasing" TRSs (Jacquemard, 1996).

Corollary 81 *Let $\mathcal{R}$ be a linear TRS, $\mathcal{A}$ be a tree automaton, α an injective abstraction such that $\mathcal{R}an(\alpha) \cap \mathcal{Q} = \emptyset$. If $\mathcal{A}^*_{\mathcal{R},\alpha}$ exists then*

$$\mathcal{L}(\mathcal{A}^*_{\mathcal{R},\alpha}) = \mathcal{R}^*(\mathcal{L}(\mathcal{A}))$$

PROOF. Left-coherence, right-coherence and coherence of Theorem 76 are trivially satisfied. □

Lemma 82 (Termination of tree automata completion) *If $\mathcal{R}an(\alpha)$ is finite then completion terminates and the tree automaton $\mathcal{A}^*_{\mathcal{R},\alpha}$ exists.*

PROOF. If $\mathcal{R}an(\alpha)$ is finite then the number of new states introduced by completion is finite. If the number of new states is finite then the set of possible transitions built on $\mathcal{F}$, $\mathcal{Q}$ and on the new states is finite and thus completion necessarily terminates. Hence, $\mathcal{A}^*_{\mathcal{R},\alpha}$ exists. □

In Section 3.3.1, we show how exact sets of reachable terms can be constructed using completion and injective finite abstraction functions α. Before that, let us show that completion can be extended to conditional TRSs.

3.1.4 Extension to the conditional case

In this section, we propose an extension of the completion algorithm for dealing with conditional term rewriting systems (CTRS for short). A natural way to compute the set of

reachable terms for CTRSs is to encode CTRSs into TRS and use the tree automata completion algorithm for TRS. However, as shown in (Feuillade and Genet, 2003; Feuillade et al., 2004), a completion algorithm adapted to the specific case of CTRS is likely to give better results in practice. This algorithm specific to the conditional case is described in this section.

A conditional term rewriting system (CTRS) over a set of ground terms $\mathcal{T}(\mathcal{F})$ is a set $\mathcal{R}$ of conditional rules $t_l \rightarrow t_r$ if $cond$, where $t_l, t_r \in \mathcal{T}(\mathcal{F}, \mathcal{X})$ and $cond$ designates a conjunction of conditions that must be checked before rewriting. Conditions are pairs of terms denoted by $c_1 \downarrow c_2$ where $c_1, c_2 \in \mathcal{T}(\mathcal{F}, \mathcal{X}), (Var(c_1) \cup Var(c_2)) \subseteq Var(t_l)$. Such a condition, called join condition, is true for a substitution σ if there exists a term $u \in \mathcal{T}(\mathcal{F})$ such that $c_1\sigma$ and $c_2\sigma$ can be both rewritten by the CTRS $\mathcal{R}$ into a same term u. Then, the rewriting $t_l\sigma \rightarrow t_r\sigma$ is said to be *enabled* and can be applied to the term $t \in \mathcal{T}(\mathcal{F})$ at position p as for a TRS. $\rightarrow_{\mathcal{R}}$ also defines a rewriting relation on $\mathcal{T}(\mathcal{F})$ and thus the set of reachable terms $\mathcal{R}^*(\mathcal{S})$ is defined as in the non-conditional case. For the following we will only consider *atomic* join conditions, and then propose an extension to conjunction of conditions in a very simple way.

For recognizing conditions in the tree automata and compute separately their value, we will need separate states as well as the property that completion builds automata where every states (not only final ones) are closed by rewriting.

We first define a rewriting relation $t \xrightarrow[\mathcal{R}]{\downarrow n} s$ meaning that to rewrite t into s, it is necessary to evaluate at most n recursive conditions (n is called the depth of the derivation in (Dershowitz et al., 1988)).

Definition 83 *For a CTRS $\mathcal{R}$ with a subset $\mathcal{R}_{nc}$ of non conditional rules, we note $\xrightarrow[\mathcal{R}]{\downarrow n}$ the relation defined by:*

- $\xrightarrow[\mathcal{R}]{\downarrow 0} = \rightarrow_{\mathcal{R}_{nc}}$
- $a \xrightarrow[\mathcal{R}]{\downarrow n+1} b \Leftrightarrow a \xrightarrow[\mathcal{R}]{\downarrow 0} b$ *or* $\exists\sigma$ *substitution*, $p \in \mathcal{P}os(a)$ *and* $(l \rightarrow r$ *if* $s \downarrow t) \in \mathcal{R}$ *such that* $a|_p = l\sigma, b = a[r\sigma]_p$ *and* $\exists u \in \mathcal{T}(\mathcal{F})$ *such that* $s\sigma \xrightarrow[\mathcal{R}]{\downarrow n}^* u$ *and* $t\sigma \xrightarrow[\mathcal{R}]{\downarrow n}^* u$.

Note that $l \rightarrow^*_{\mathcal{R}} r$ means that $\exists n \in \mathbb{N}$ s.t. $l \xrightarrow[\mathcal{R}]{\downarrow n}^* r$. ◇

Let $\mathcal{A}_0$ be the tree automaton to be completed using the left-linear CTRS $\mathcal{R}$. Let us consider the following algorithm, where we complete at the ith step the automaton $\mathcal{A}^i_{\mathcal{R}} = \langle \mathcal{F}, \mathcal{Q}_i, \mathcal{Q}_f, \Delta_i \rangle$ to an automaton $\mathcal{A}^{i+1}_{\mathcal{R}}$. The set of state Q_i is partitioned into three set of states: $Q_0 \cup Q_{i,new} \cup Q_{i,cond}$. Q_0 is the set of states of $\mathcal{A}_0$, $Q_{i,new}$ is a set of states produced by transition normalization and indexed by naturals, $Q_{i,cond}$ is a set of conditional states indexed by terms of $\mathcal{T}(\mathcal{F}, Q_i)$. Let α be an abstraction function. We use the following algorithm :

1. from $\mathcal{A}_i = \langle \mathcal{F}, \mathcal{Q}_i, \mathcal{Q}_f, \Delta_i \rangle$, the ith step of completion, we compute the automaton $\mathcal{A}_{i+1} = \langle \mathcal{F}, \mathcal{Q}_{i+1}, \mathcal{Q}_f, \Delta_{i+1} \rangle$ with the initialization: $\mathcal{Q}_{i+1} = \mathcal{Q}_i, \Delta_{i+1} = \Delta_i$.

2. Let us consider each *critical pair* without considering the condition of the rule. A pair (q, r) of $\mathcal{Q} \times \mathcal{T}(\mathcal{F})$ is said to be critical for a rule either non conditional $l \to r$, or conditional $l \to r$ if $c_1 \downarrow c_2$ where $\sigma \in \Sigma(\mathcal{Q}, \mathcal{X})$ is a regular language substitution $\sigma = \{x_1 \mapsto q_{i_1}, x_2 \mapsto q_{i_2}, \ldots, x_n \mapsto q_{i_n}\}$, where $\{x_1, x_2, \ldots, x_n\} = var(l)$, if $l\sigma \to^*_{\Delta_i} q$ and $r\sigma \not\to^*_{\Delta_i} q$.

3. for all of these critical pairs, the rule is either:

   - a conditional one: $l \to r$ if $c_1 \downarrow c_2$. There are two possibilities
     - there is no state indexed by $c_1\sigma$ or $c_2\sigma$ in the conditional subset of states of $\mathcal{Q}_i$ ($q_{c_1\sigma} \notin \mathcal{Q}_{i,cond}$ or $q_{c_2\sigma} \notin \mathcal{Q}_{i,cond}$), then we create these two states (or the one missing) and we add to the automaton $\mathcal{A}_{i+1}$ the following transitions :

       $$Norm_\alpha(c_1\sigma \to q_{c_1\sigma}) \cup Norm_\alpha(c_2\sigma \to q_{c_2\sigma})$$

     - there exist two states $q_{c_1\sigma}$ and $q_{c_2\sigma}$ in $\mathcal{Q}_i$. We calculate $\mathcal{L}(\mathcal{A}_i, q_{c_1\sigma}) \cap \mathcal{L}(\mathcal{A}_i, q_{c_2\sigma})$. If this set is empty, the condition is, for this completion step, considered as false. If it is not empty, then the condition is true and we go on processing the critical pair as if the rule were not conditional.
   - a non conditional one (or it is conditional and the condition has been found true in the previous step), then we add to the automaton the transitions $Norm_\alpha(r\sigma \to q)$.

4. the new automaton $\mathcal{A}_{i+1} = \langle \mathcal{F}, \mathcal{Q}_{i+1}, \mathcal{Q}_{f,i+1}, \Delta_{i+1} \rangle$ is the result of one step of completion of $\mathcal{A}_i$.

If there exists $i \in \mathbb{N}$ such that $\mathcal{A}_i = \mathcal{A}_{i+1}$, then $\mathcal{A}_i$ is the result. Each time we add a transition to the automaton, we have to normalize it with new states (indexed by naturals and added in $Q_{i,new}$) and then we have the opportunity to make an approximation in order to limit the number of new states created for the normalization. As in the non conditional case, this completion may not have a fixed point: we may produce infinitely many new states. However, approximation techniques similar to those of Section 3.1.2 apply: let Q_{cond} be the set of new states $q_{c_1\sigma}$ and $q_{c_1\sigma}$ produced by conditions, $\mathcal{Q}_{new}$ the set of new states used to normalize the transitions, one may restrict in any way the set $\mathcal{Q}_{new}$ to force completion to terminate. Note that there is no need to limit the number of states of Q_{cond}, since the number of possible conditions c_1, c_2 is finite and the number of possible σ is finite if $\mathcal{Q}_{new}$ is.

Theorem 84 *Let A_0 be a tree automaton such that $\mathcal{L}(\mathcal{A}_0) \supseteq \mathcal{S}$ and $\mathcal{R}$ a left linear CTRS. If $\mathcal{A}'$ is the result of the completion of $\mathcal{A}_0$ w.r.t $\mathcal{R}$, then $\mathcal{L}(\mathcal{A}')$ is closed with respect to $\mathcal{R}$ and $\mathcal{R}^*(\mathcal{S}) \subseteq \mathcal{R}^*(\mathcal{L}(\mathcal{A}_0)) \subseteq \mathcal{L}(\mathcal{A}')$*

PROOF. Let $\mathcal{A}' = \langle \mathcal{F}, \mathcal{Q}', \mathcal{Q}_f, \Delta' \rangle$. We prove that $\forall t \in \mathcal{T}(\mathcal{F})$ s.t. $\exists q \in \mathcal{Q}', t \in \mathcal{L}(\mathcal{A}', q)$, $\forall u \in \mathcal{T}(\mathcal{F})$ s.t. $t \to^*_{\mathcal{R}} u$, we have $u \in \mathcal{L}(\mathcal{A}', q)$. We prove by induction on n that $\forall n \in \mathbb{N}, q \in \mathcal{Q}', t \in \mathcal{L}(\mathcal{A}', q)$, u s.t. $t \xrightarrow{\downarrow n}{}^*_{\mathcal{R}} u$, then $u \in \mathcal{L}(\mathcal{A}', q)$

- If $q \in \mathcal{Q}', t \in \mathcal{L}(\mathcal{A}', q)$, and $t \xrightarrow[\mathcal{R}]{\downarrow 0}^* u$ then we trivially have $u \in \mathcal{L}(\mathcal{A}', q)$. Indeed, $\xrightarrow[\mathcal{R}]{\downarrow 0}^*$ means that we consider the subset of non conditional rules of $\mathcal{R}$ and then the proof follows from theorem 67 for the automaton $\mathcal{A}'$ with q as final state.

- now suppose that for a given n: $\forall k \leq n, t \xrightarrow[\mathcal{R}]{\downarrow k}^* u$ and $t \rightarrow^*_{\Delta'} q \Rightarrow u \rightarrow^*_{\Delta'} q$. We want to show that:
$$t \xrightarrow[\mathcal{R}]{\downarrow n+1}^* u \text{ and } t \rightarrow^*_{\Delta'} q \Rightarrow u \rightarrow^*_{\Delta'} q$$
$t \xrightarrow[\mathcal{R}]{\downarrow n+1}^* u$ means that exists $\{t_1, t_2, \ldots, t_j\} \subseteq \mathcal{T}(\mathcal{F})$ such that
$$t_0 = t \xrightarrow[\mathcal{R}]{\downarrow n+1} t_1 \xrightarrow[\mathcal{R}]{\downarrow n+1} t_2 \xrightarrow[\mathcal{R}]{\downarrow n+1} \ldots \xrightarrow[\mathcal{R}]{\downarrow n+1} t_{j-1} \xrightarrow[\mathcal{R}]{\downarrow n+1} t_j = u$$
Now we show that for every t_i, if $t_i \rightarrow^*_{\Delta'} q$ then $t_{i+1} \rightarrow^*_{\Delta'} q$, this leads to two cases:
  - $t_i \xrightarrow[\mathcal{R}]{\downarrow n} t_{i+1}$, then using the induction hypothesis, $t_{i+1} \rightarrow^*_{\Delta'} q$.
  - $t_i \not\xrightarrow[\mathcal{R}]{\downarrow n} t_{i+1}$ and $t_i \xrightarrow[\mathcal{R}]{\downarrow n+1} t_{i+1}$, so there exists a rule $(k)\, l \rightarrow r$ if $c_1 \downarrow c_2 \in \mathcal{R}$ a closed context $C[]$, and a substitution σ such that:
$$t_i = C[l\sigma] \rightarrow_{\mathcal{R}} C[r\sigma] = t_{i+1} \text{ if } c_1\sigma \downarrow c_2\sigma$$
$$\text{and } \exists c \text{ s.t. } c_1\sigma \xrightarrow[\mathcal{R}]{\downarrow n}^* c, \text{ and } c_2\sigma \xrightarrow[\mathcal{R}]{\downarrow n}^* c$$
Since no critical pair between $\mathcal{R}$ and Δ' exists because the automaton is a fixed point for the completion, we necessarily have that $\exists q_{c_1\sigma}, q_{c_2\sigma} \in \mathcal{Q}'$. Thus, we have:
$$c_1\sigma \xrightarrow[\mathcal{R}]{\downarrow n}^* c \quad \text{and } c_1\sigma \rightarrow^*_{\Delta'} q_{c_1\sigma}$$
$$c_2\sigma \xrightarrow[\mathcal{R}]{\downarrow n}^* c \quad \text{and } c_1\sigma \rightarrow^*_{\Delta'} q_{c_2\sigma}$$
The induction hypothesis leads to $c \in \mathcal{L}(\mathcal{A}', q_{c_1\sigma})$ and $c \in \mathcal{L}(\mathcal{A}', q_{c_2\sigma})$. Consequently, since $t_i \rightarrow^*_{\Delta'} q$, the condition $\mathcal{L}(\mathcal{A}', q_{c_1,\sigma}) \cap \mathcal{L}(\mathcal{A}', q_{c_2,\sigma}) \neq \emptyset$ is true, and $\mathcal{A}$ is a fixed point for the completion for automaton $\mathcal{A}$, we necessarily have $t_{i+1} \rightarrow^*_{\Delta'} q$.

  We have $t = t_0 \in \mathcal{L}(\mathcal{A}', q)$, so by induction $\forall i \leq j, t_i \in \mathcal{L}(\mathcal{A}', q)$, in particular $u = t_j$.

We get the result that $t \xrightarrow[\mathcal{R}]{\downarrow n+1}^* u$ and $t \rightarrow^*_{\Delta} q$ implies $u \rightarrow^*_{\Delta} q$

So $\forall n \in \mathbb{N}, t \xrightarrow[\mathcal{R}]{\downarrow n}^* u$ and $t \rightarrow^*_{\Delta'} q$ implies $u \rightarrow^*_{\Delta'} q$, then $t \rightarrow^*_{\mathcal{R}} u$ and $t \rightarrow^*_{\Delta'} q$ implies $u \rightarrow^*_{\Delta'} q$. This leads us to $\forall q \in \mathcal{Q}', \mathcal{L}(\mathcal{A}', q)$ is closed under rewriting by $\mathcal{R}$, in particular for $q \in \mathcal{Q}_f$, thus $\mathcal{L}(\mathcal{A}')$ is closed under rewriting by $\mathcal{R}$. Since completion is

incremental, we have the inequalities $\Delta \subseteq \Delta'$ and thus $\mathcal{S} \subseteq \mathcal{L}(\mathcal{A}_0) \subseteq \mathcal{L}(\mathcal{A}')$, and finally $\mathcal{R}^*(\mathcal{S}) \subseteq \mathcal{L}(\mathcal{A}')$. □

In order to compute the completion for a set of rules including rules with conjunction of conditions, one can modify the algorithm in the following way: when such a rule is involved in a critical pair, create a pair of new states for each atomic condition. Later, if the same critical pair is found, the rule is enabled for the considered substitution if each related atomic condition is satisfied (according to the emptiness checking of the intersection of the conditional state languages). Then we have to normalize the right-hand term as in the non conditional case.

3.2 Equational Tree Automata Completion

In this section, we propose a refinement of the tree automata completion called *equational tree automata completion*. The objective of the refinement is double. First, we aim at replacing normalization rules by equations in order to ease the definition of approximations and also to authorize the evaluation of their precision. This is the subject of Section 3.2.1. Second, this new algorithm permits to gain precision in the completed tree automaton by preserving the rewriting graph of the reachable terms. This is based on a different way of solving critical pairs which is presented in Section 3.2.3.

3.2.1 Simplification of Tree Automata by Equations

In this section, we define the *simplification* of tree automata $\mathcal{A}$ w.r.t. a set of equations E. This operation consists in finding E-equivalent terms recognized in $\mathcal{A}$ by different states and then by merging those states together. The merging of states is performed using renaming of a state in a tree automaton.

Definition 85 (Renaming of a state in a tree automaton) *Let $\mathcal{Q}, \mathcal{Q}'$ be set of states, $\mathcal{A} = \langle \mathcal{F}, \mathcal{Q}, \mathcal{Q}_f, \Delta \rangle$ be a tree automaton, and α a function $\alpha : \mathcal{Q} \mapsto \mathcal{Q}'$. We denote by $\mathcal{A}\alpha$ the tree automaton where every occurrence of q is replaced by $\alpha(q)$ in $\mathcal{Q}$, $\mathcal{Q}_f$ and in every left and right-hand side of every transition of Δ.* ◇

If there exists a bijection α such that $\mathcal{A} = \mathcal{A}'\alpha$ then $\mathcal{A}$ and $\mathcal{A}'$ are said to be *equivalent modulo renaming*. The following lemma shows that every term recognized in $\mathcal{A}$ is also recognized in the renamed version of $\mathcal{A}$.

Lemma 86 *Let $\mathcal{A}, \mathcal{A}'$ be tree automata and q, q_a, q_b states of $\mathcal{A}$ such that $\mathcal{A}' = \mathcal{A}\{q_a \mapsto q_b\}$. For all term $t \in \mathcal{T}(\mathcal{F} \cup \mathcal{Q})$:*

- *if $t \rightarrow_{\mathcal{A}}^{\not\epsilon *} q$ then $t\{q_a \mapsto q_b\} \rightarrow_{\mathcal{A}'}^{\not\epsilon *} q\{q_a \mapsto q_b\}$*
- *if $t \rightarrow_{\mathcal{A}}^{*} q$ then $t\{q_a \mapsto q_b\} \rightarrow_{\mathcal{A}'}^{*} q\{q_a \mapsto q_b\}$*

PROOF. We detail here the proof for $t \rightarrow_{\mathcal{A}}^{*} q \implies t \rightarrow_{\mathcal{A}'}^{*} q\{q_a \mapsto q_b\}$. The proof for the $\rightarrow_{\mathcal{A}}^{\not\epsilon *}$ case is simpler. We proceed by induction on the height of term t:

- if t is of height 0 then t is a state q_1 such that $q_1 \rightarrow_{\mathcal{A}} q_2 \rightarrow_{\mathcal{A}} \ldots \rightarrow_{\mathcal{A}} q_n = q$. If $\forall i = 1 \ldots n : q_i \neq q_a$ then the same derivation can be performed within $\mathcal{A}'$ since all transitions $\forall i = 1 \ldots n-1 : q_i \rightarrow_{\mathcal{A}} q_{i+1}$ are part of $\mathcal{A}'$, and thus $q_1 \rightarrow^*_{\mathcal{A}'} q_n = q$. If $\exists k = 1 \ldots n : q_k = q_a$, we have $q_1 \rightarrow_{\mathcal{A}} q_2 \rightarrow_{\mathcal{A}} \ldots \rightarrow_{\mathcal{A}} q_{k-1} \rightarrow_{\mathcal{A}} q_a \rightarrow_{\mathcal{A}} q_{k+1} \rightarrow_{\mathcal{A}} \ldots \rightarrow_{\mathcal{A}} q_n$. In $\mathcal{A}'$ transitions $q_{k-1} \rightarrow q_a$ and $q_a \rightarrow q_{k+1}$ will respectively be replaced by $q_{k-1} \rightarrow q_b$ and $q_b \rightarrow q_{k+1}$. Hence we have $q_1 \rightarrow_{\mathcal{A}'} q_2 \rightarrow_{\mathcal{A}'} \ldots \rightarrow_{\mathcal{A}'} q_{k-1} \rightarrow_{\mathcal{A}'} q_b \rightarrow_{\mathcal{A}'} q_{k+1} \rightarrow_{\mathcal{A}'} \ldots \rightarrow_{\mathcal{A}'} q_n$, or $q_1 \rightarrow^*_{\mathcal{A}'} q_n = q$.

- now, we assume that the property is true for terms of height lesser or equal to n. Let us prove the property for the term $t = f(t_1, \ldots, t_n)$ of height lesser or equal to $n+1$. We know that $f(t_1, \ldots, t_n) \rightarrow_{\mathcal{A}}{}^* q$. By the construction of tree automaton derivation, we can deduce that there exists states $q_1, \ldots, q_n$ such that $f(t_1, \ldots, t_n) \rightarrow_{\mathcal{A}}{}^* f(q_1, \ldots, q_n) \rightarrow_{\mathcal{A}}{}^* q$. Applying the induction hypothesis on $t_1 \rightarrow_{\mathcal{A}}{}^* q_1, \ldots, t_n \rightarrow_{\mathcal{A}}{}^* q_n$ and $f(q_1, \ldots, q_n) \rightarrow_{\mathcal{A}}{}^* q$, we obtain that $t_1\{q_a \mapsto q_b\} \rightarrow^*_{\mathcal{A}'} q_1\{q_a \mapsto q_b\}, \ldots, t_n\{q_a \mapsto q_b\} \rightarrow^*_{\mathcal{A}'} q_n\{q_a \mapsto q_b\}$ and $f(q_1\{q_a \mapsto q_b\}, \ldots, q_n\{q_a \mapsto q_b\}) \rightarrow^*_{\mathcal{A}'} q\{q_a \mapsto q_b\}$. Thus, $f(t_1, \ldots, t_n)\{q_a \mapsto q_b\} \rightarrow^*_{\mathcal{A}'} f(q_1, \ldots, q_n)\{q_a \mapsto q_b\} \rightarrow^*_{\mathcal{A}'} q\{q_a \mapsto q_b\}$.

□

We use renamings for merging states in a tree automaton. For instance, a tree automaton $\mathcal{A}$ where q_1 and q_2 are merged is $\mathcal{A}\{q_2 \mapsto q_1\}$, i.e. a tree automaton where every occurrence of q_2 has been replaced by q_1. Now we define the *simplification relation* which merges states in a tree automaton according to an equation.

Definition 87 (Simplification relation) *Let $\mathcal{A} = \langle \mathcal{F}, \mathcal{Q}, \mathcal{Q}_f, \Delta \rangle$ be a tree automaton and E be a set of linear equations. For $s = t \in E$, $\sigma \in \Sigma(\mathcal{Q}, \mathcal{X})$, $q_a, q_b \in \mathcal{Q}$ such that $s\sigma \rightarrow^{\not\epsilon *}_{\mathcal{A}} q_a$, $t\sigma \rightarrow^{\not\epsilon *}_{\mathcal{A}} q_b$, i.e.*

$$\begin{array}{ccc} s\sigma & \underset{E}{=\!=} & t\sigma \\ {\scriptstyle \mathcal{A},\not\epsilon}\downarrow{\scriptstyle *} & & {\scriptstyle *}\downarrow{\scriptstyle \mathcal{A},\not\epsilon} \\ q_a & & q_b \end{array}$$

and $q_a \neq q_b$ then $\mathcal{A}$ can be simplified *into $\mathcal{A}' = \mathcal{A}\{q_b \mapsto q_a\}$, denoted by $\mathcal{A} \rightsquigarrow_E \mathcal{A}'$.*

◇ ◇

Lemma 88 *For all tree automata $\mathcal{A}, \mathcal{A}'$, all set of equations E and all states q, q_a, q_b of $\mathcal{A}$ such that $\mathcal{A} \rightsquigarrow_E \mathcal{A}'$ and $\mathcal{A}' = \mathcal{A}\{q_a \mapsto q_b\}$, we have $\mathcal{L}(\mathcal{A}, q) \subseteq \mathcal{L}(\mathcal{A}\{q_a \mapsto q_b\}, q\{q_a \mapsto q_b\})$.*

PROOF. Direct consequence of Lemma 86 □

To build approximations, our objective is to apply repeatedly simplification on a tree automaton up to a normal form. However, simplification is non deterministic: at each step of simplification several equations may be used and on different couples of states. Hence, it is important to guarantee that whatever the order of simplification steps performed the

normal form is unique. First, we show that the simplification relation is well-founded. Then that it is locally confluent. From those two lemmas, we can get the confluence and uniqueness of normal forms.

Lemma 89 *The simplification relation $\rightsquigarrow_E$ is well-founded.*

PROOF. Each step of simplification $\mathcal{A} \rightsquigarrow_E \mathcal{A}'$ replace every occurrence of a state q_2 by q_1 in $\mathcal{A}$ to obtain $\mathcal{A}'$. Hence, each step of simplification strictly decreases the number of states in the tree automaton, which is bounded by 1. □

Definition 90 *For tree automata $\mathcal{A}, \mathcal{A}'$, we note $\mathcal{A} \rightsquigarrow_E^! \mathcal{A}'$ if $\mathcal{A} \rightsquigarrow_E^* \mathcal{A}'$ and $\mathcal{A}'$ is normal form w.r.t. $\rightsquigarrow_E$, i.e. $\forall \mathcal{A}'' : \mathcal{A}' \not\rightsquigarrow_E \mathcal{A}''$.* ◇

Lemma 91 (Local confluence) *The relation $\rightsquigarrow_E$ is locally confluent modulo a bijective renaming of states.*

PROOF. We prove here that for all tree automata $\mathcal{A}, \mathcal{A}_1, \mathcal{A}_2$ if $\mathcal{A} \rightsquigarrow \mathcal{A}_1$ and $\mathcal{A} \rightsquigarrow \mathcal{A}_2$ then there exists a two tree automata $\mathcal{A}'_1$ such that $\mathcal{A}_1 \rightsquigarrow_E^* \mathcal{A}'_1$, $\mathcal{A}_2 \rightsquigarrow_E^* \mathcal{A}'_2$ and $\mathcal{A}'_1$ and $\mathcal{A}'_2$ are equivalent modulo a bijective renaming of their states. Assume that $\mathcal{A}_1$ (resp. $\mathcal{A}_2$) was obtained from $\mathcal{A}$ using the equation $l_1 = r_1$ (resp. $l_2 = r_2$) such that $l_1\sigma_1 \rightarrow_{\mathcal{A}}^{\not\epsilon *} q_a$ and $r_1\sigma_1 \rightarrow_{\mathcal{A}}^{\not\epsilon *} q_b$ (resp. $l_2\sigma_2 \rightarrow_{\mathcal{A}}^{\not\epsilon *} q_c$ and $r_2\sigma_2 \rightarrow_{\mathcal{A}}^{\not\epsilon *} q_d$). Hence $\mathcal{A}_1 = \mathcal{A}\{q_a \mapsto q_b\}$ and $\mathcal{A}_2 = \mathcal{A}\{q_c \mapsto q_d\}$. Using this and Lemma 86, we get that there exists substitutions $\sigma'_1 = \sigma_1\{q_c \mapsto q_d\}$ and $\sigma'_2 = \sigma_2\{q_a \mapsto q_b\}$ such that $l_1\sigma'_1 \rightarrow_{\mathcal{A}_2}^{\not\epsilon *} q_a\{q_c \mapsto q_d\}$ and $r_1\sigma'_1 \rightarrow_{\mathcal{A}_2}^{\not\epsilon *} q_b\{q_c \mapsto q_d\}$ (resp. $l_2\sigma'_2 \rightarrow_{\mathcal{A}_1}^{\not\epsilon *} q_c\{q_a \mapsto q_b\}$ and $r_2\sigma'_2 \rightarrow_{\mathcal{A}_1}^{\not\epsilon *} q_d\{q_a \mapsto q_b\}$). By Definition87, we know that we necessarily have $q_a \neq q_b$ and $q_c \neq q_d$. Then we proceed by case study on the possibly equivalent states:

- if $q_a = q_c$, we are going to study two different cases: $q_b = q_d$ and $q_b \neq q_d$:
  - if $q_b = q_d$ then we trivially have $\mathcal{A}_1 = \mathcal{A}\{q_a \mapsto q_b\} = \mathcal{A}_2$. Thus, $\mathcal{A}'_1 = \mathcal{A}_1 = \mathcal{A}_2 = \mathcal{A}'_2$;
  - if $q_b \neq q_d$ then, from $q_a = q_c$, we get that $\mathcal{A}_1 = \mathcal{A}\{q_a \mapsto q_b\}$ and $\mathcal{A}_2 = \mathcal{A}\{q_a \mapsto q_d\}$. Besides this, as we have seen above, we know that there exists a substitution σ'_1 such that $l_1\sigma'_1 \rightarrow_{\mathcal{A}_2}^{\not\epsilon *} q_a\{q_a \mapsto q_d\}$ and $r_1\sigma'_1 \rightarrow_{\mathcal{A}_2}^{\not\epsilon *} q_b\{q_a \mapsto q_d\}$. Since we know from the beginning that $q_a \neq q_b$, we get that $l_1\sigma'_1 \rightarrow_{\mathcal{A}_2}^{\not\epsilon *} q_d$ and $r_1\sigma'_1 \rightarrow_{\mathcal{A}_2}^{\not\epsilon *} q_b$. Since we are in the case where $q_b \neq q_d$, we know that simplification applies. Hence there exists a tree automaton $\mathcal{A}'_2 = \mathcal{A}_2\{q_d \mapsto q_b\}$ such that $\mathcal{A}_2 \rightsquigarrow_E \mathcal{A}'_2$. We can apply the same reasoning on $\mathcal{A}_1$. Remember that $l_2\sigma'_2 \rightarrow_{\mathcal{A}_1}^{\not\epsilon *} q_a\{q_a \mapsto q_b\}$ and $r_2\sigma'_2 \rightarrow_{\mathcal{A}_1}^{\not\epsilon *} q_d\{q_a \mapsto q_b\}$. Besides to this we know that $q_d \neq q_a$ because otherwise we would have $q_d = q_a = q_C$ which is a contradiction with $q_c \neq q_d$. Hence renamings on states can be applied and we obtain: $l_2\sigma'_2 \rightarrow_{\mathcal{A}_1}^{\not\epsilon *} q_b$ and $r_2\sigma'_2 \rightarrow_{\mathcal{A}_1}^{\not\epsilon *} q_d$. Like in the previous case, this implies that there exists a tree automaton $\mathcal{A}'_1$ such that $\mathcal{A}_1 \rightsquigarrow_E \mathcal{A}'_1$ and $\mathcal{A}'_1 = \mathcal{A}_1\{q_b \mapsto q_d\}$. What remains to be proved is that $\mathcal{A}'_1$ and $\mathcal{A}'_2$ are equivalent modulo renaming.

$$\begin{array}{rcl}
\mathcal{A}'_1 & = & \mathcal{A}'_2 \\
\mathcal{A}_1\{q_b \mapsto q_d\} & = & \mathcal{A}_2\{q_d \mapsto q_b\} \\
(\mathcal{A}\{q_a \mapsto q_b\})\{q_b \mapsto q_d\} & = & (\mathcal{A}\{q_a \mapsto q_d\})\{q_d \mapsto q_b\} \\
\mathcal{A}\{q_a \mapsto q_d, q_b \mapsto q_d\} & = & \mathcal{A}\{q_a \mapsto q_b, q_d \mapsto q_b\}
\end{array}$$

Hence, $\mathcal{A}'_1$ is $\mathcal{A}$ where states q_a, q_b, q_d are all merged together in q_d and $\mathcal{A}'_2$ is $\mathcal{A}$ where q_a, q_b, q_d are merged in q_b. We can thus conclude that $\mathcal{A}'_1$ can be obtained from $\mathcal{A}'_2$ by simply renaming q_b in q_d. This renaming is bijective since q_d does not occur in $\mathcal{A}'_2$.

- if $q_a \neq q_c$ and $q_b = q_d$ then $\mathcal{A}_1 = \mathcal{A}\{q_a \mapsto q_b\}$ and $\mathcal{A}_2 = \{q_c \mapsto q_b\}$. Like in the previous case, we can use the fact that $l_1\sigma'_1 \rightarrow^{\not{c}*}_{\mathcal{A}_2} q_a\{q_c \mapsto q_b\}$ and $r_1\sigma'_1 \rightarrow^{\not{c}*}_{\mathcal{A}_2} q_b\{q_c \mapsto q_b\}$ to deduce that $l_1\sigma'_1 \rightarrow^{\not{c}*}_{\mathcal{A}_2} q_a$ and $r_1\sigma'_1 \rightarrow^{\not{c}*}_{\mathcal{A}_2} q_b$. The renamings can be simplified in that way because, in the case we are studying $q_a \neq q_c$ and we also have $q_b \neq q_c$ because, otherwise, we would have $q_c = q_b = q_d$ which is a contradiction with $q_c \neq q_d$. From previous application of rule $l_1 = r_1$ we get that $\mathcal{A}_2 \rightsquigarrow_E \mathcal{A}'_2 = \mathcal{A}_2\{q_a \mapsto q_b\}$. From the symmetrical property, i.e. $l_2\sigma'_2 \rightarrow^{\not{c}*}_{\mathcal{A}_1} q_c\{q_a \mapsto q_b\}$ and $r_2\sigma'_2 \rightarrow^{\not{c}*}_{\mathcal{A}_1} q_b\{q_a \mapsto q_b\}$, and the fact that $q_a \neq q_c$ and $q_a \neq q_b$, we obtain: $l_2\sigma'_2 \rightarrow^{\not{c}*}_{\mathcal{A}_1} q_c$ and $r_2\sigma'_2 \rightarrow^{\not{c}*}_{\mathcal{A}_1} q_b$. Hence, $\mathcal{A}_1 \rightsquigarrow_E \mathcal{A}'_1 = \mathcal{A}_1\{q_c \rightarrow q_b\}$. Like in the previous case, we have:

$$\begin{array}{rcl}
\mathcal{A}'_1 & = & \mathcal{A}'_2 \\
\mathcal{A}_1\{q_c \mapsto q_b\} & = & \mathcal{A}_2\{q_a \mapsto q_b\} \\
(\mathcal{A}\{q_a \mapsto q_b\})\{q_c \mapsto q_b\} & = & (\mathcal{A}\{q_c \mapsto q_b\})\{q_a \mapsto q_b\} \\
\mathcal{A}\{q_a \mapsto q_b, q_c \mapsto q_b\} & = & \mathcal{A}\{q_c \mapsto q_b, q_a \mapsto q_b\}
\end{array}$$

Here, we directly have $\mathcal{A}'_1 = \mathcal{A}'_2$.

- if $q_a \neq q_c$, $q_d \neq q_b$ and $q_b = q_c$, then $\mathcal{A}_1 = \mathcal{A}\{q_a \mapsto q_b\}$ and $\mathcal{A}_2 = \{q_b \mapsto q_d\}$.
  - if $q_a = q_d$ then $\mathcal{A}_1 = \mathcal{A}\{q_a \mapsto q_b\}$ and $\mathcal{A}_2 = \{q_b \mapsto q_a\}$. Then we directly have $\mathcal{A}'_1 = \mathcal{A}_1$ and $\mathcal{A}'_2 = \mathcal{A}_2$ since $\mathcal{A}_1$ and $\mathcal{A}_2$ are equivalent modulo a bijective renaming. Indeed, $\mathcal{A}_1$ can be obtained from $\mathcal{A}_2$ by simply renaming q_a in q_b. This renaming is bijective since q_b does not occur in $\mathcal{A}_2$.
  - if $q_a \neq q_d$, then from $l_1\sigma'_1 \rightarrow^{\not{c}*}_{\mathcal{A}_2} q_a\{q_c \mapsto q_d\}$ and $r_1\sigma'_1 \rightarrow^{\not{c}*}_{\mathcal{A}_2} q_b\{q_b \mapsto q_d\}$ we get that simplification applies and $\mathcal{A}_2 \rightsquigarrow_E \mathcal{A}'_2 = \mathcal{A}_2\{q_a \mapsto q_d\}$. For $\mathcal{A}_1$ we get that $l_2\sigma'_2 \rightarrow^{\not{c}*}_{\mathcal{A}_1} q_b\{q_a \mapsto q_b\}$ and $r_2\sigma'_2 \rightarrow^{\not{c}*}_{\mathcal{A}_1} q_d\{q_a \mapsto q_b\}$ and thus $A_1 \rightsquigarrow_E \mathcal{A}'_1 = \mathcal{A}_1\{q_b \mapsto q_d\}$.

$$\begin{array}{rcl} \mathcal{A}'_1 & = & \mathcal{A}'_2 \\ \mathcal{A}_1\{q_b \mapsto q_d\} & = & \mathcal{A}_2\{q_a \mapsto q_d\} \\ (\mathcal{A}\{q_a \mapsto q_b\})\{q_b \mapsto q_d\} & = & (\mathcal{A}\{q_b \mapsto q_d\})\{q_a \mapsto q_d\} \\ \mathcal{A}\{q_a \mapsto q_d, q_b \mapsto q_d\} & = & \mathcal{A}\{q_b \mapsto q_d, q_a \mapsto q_d\} \end{array}$$

Here, we also directly have $\mathcal{A}'_1 = \mathcal{A}'_2$.

- the case where $q_a \neq q_c$, $q_d \neq q_b$ and $q_a = q_d$ is symmetrical to the previous one with $q_b = q_c$ and can thus be treated in the same way.
- In the remaining case, we know that $q_a \neq q_c$, $q_d \neq q_b$, $q_b \neq q_c$ and $q_a \neq q_d$. Furthermore, as mentioned before, we have the general hypothesis saying that $q_a \neq q_b$ and $q_c \neq q_d$. Hence, in that case, we know that q_a, q_b, q_c, q_d are all different. As before, from the usual property saying that $l_1\sigma'_1 \rightarrow^{\not{\epsilon}*}_{\mathcal{A}_2} q_a\{q_c \mapsto q_d\}$ and $r_1\sigma'_1 \rightarrow^{\not{\epsilon}*}_{\mathcal{A}_2} q_b\{q_c \mapsto q_d\}$, we get that there exists $\mathcal{A}'_2$ such that $\mathcal{A}_2 \rightsquigarrow_E \mathcal{A}'_2\{q_a \mapsto q_b\}$. Symmetrically, there exists $\mathcal{A}'_1$ such that $\mathcal{A}_1 \rightsquigarrow_E \mathcal{A}'_1 = \mathcal{A}'_1\{q_c \mapsto q_d\}$. Finally, we have to prove that:

$$\begin{array}{rcl} \mathcal{A}'_1 & = & \mathcal{A}'_2 \\ \mathcal{A}_1\{q_c \mapsto q_d\} & = & \mathcal{A}_2\{q_a \mapsto q_b\} \\ (\mathcal{A}\{q_a \mapsto q_b\})\{q_c \mapsto q_d\} & = & (\mathcal{A}\{q_c \mapsto q_d\})\{q_a \mapsto q_b\} \end{array}$$

  Then, we can remark that, since q_a, q_b, q_c, q_d are all different, the ordering of renaming has no effect, i.e. $(\mathcal{A}\{q_a \mapsto q_b\})\{q_c \mapsto q_d\} = \mathcal{A}\{q_a \mapsto q_b, q_c \mapsto q_d\} = (\mathcal{A}\{q_c \mapsto q_d\})\{q_a \mapsto q_b\}$. Thus $\mathcal{A}'_1 = \mathcal{A}'_2$.

□

Thanks to the previous lemma and termination of $\rightsquigarrow_E$, we can prove that $\rightsquigarrow_E$ is confluent and that normal forms of $\rightsquigarrow_E$ are unique modulo renaming.

Theorem 92 (Canonical Simplified Tree Automata) *Let $\mathcal{A}, \mathcal{A}'_1, \mathcal{A}'_2$ be tree automata and E be a set of linear equations such that $\mathcal{A} \rightsquigarrow^!_E \mathcal{A}'_1$ and $\mathcal{A} \rightsquigarrow^!_E \mathcal{A}'_2$ then $\mathcal{A}'_1$ and $\mathcal{A}'_2$ are equivalent modulo a bijective renaming.*

PROOF. Since $\rightsquigarrow_E$ is locally confluent and terminating, it is confluent. This is a general property of abstract reduction systems (Baader and Nipkow, 1998). Moreover, by termination and confluence, we get the unicity of normal forms modulo a bijective renaming. □

3.2.2 $\mathcal{R}/E$-coherent Tree Automata

Now, we define more precisely the notion of $\mathcal{R}/E$-coherence of tree automata which is a key notion to prove the precision of approximations performed using equations. In the main theorem of this part, we also show that simplification relation preserves $\mathcal{R}/E$-coherence.

Definition 93 (Coherent automaton) *Let $\mathcal{A} = \langle \mathcal{F}, \mathcal{Q}, \mathcal{Q}_f, \Delta \rangle$ a tree automaton, $\mathcal{R}$ a TRS and E a set of equations. The automaton $\mathcal{A}$ is said to be* $\mathcal{R}/E$-coherent *if* $\forall q \in \mathcal{Q} : \exists s \in \mathcal{T}(\mathcal{F}) :$

$$s \rightarrow_{\mathcal{A}}^{\not/*} q \wedge [\forall t \in \mathcal{T}(\mathcal{F}) : (t \rightarrow_{\mathcal{A}}^{\not/*} q \implies s =_E t) \wedge (t \rightarrow_{\mathcal{A}}{}^* q \implies s \rightarrow_{\mathcal{R}/E}^* t)].$$

$\diamond$

In the following, any term s such that $s \rightarrow_{\mathcal{A}}^{\not/*} q$ is called a *representative* of q in $\mathcal{A}$ and is denoted by $rep(q)$. Another property enjoyed by $\mathcal{R}/E$-coherent tree automata is that they are *filled*. We now define this property and prove that it is true for $\mathcal{R}/E$-coherent tree automata.

Definition 94 *A state q of a tree automaton $\mathcal{A}$ is* filled *if there exists at least one term $t \in \mathcal{T}(\mathcal{F})$ such that $t \rightarrow_{\mathcal{A}}^{\not/*} q$. A tree automaton is filled if all its states are.* $\diamond$

Lemma 95 *Any $\mathcal{R}/E$-coherent tree automaton is filled.*

PROOF. This is clear since, for every state q of a coherent tree automaton $\mathcal{A}$, there exists a representative s such that $s \rightarrow_{\mathcal{A}}^{\not/*} q$. $\square$

Now we prove a lemma on the substitutions obtained by matching a linear term on a filled tree automaton. This lemma is crucial for the next theorem that shows that simplification preserves $\mathcal{R}/E$-coherence.

Lemma 96 *Let $\mathcal{A}$ be a filled tree automaton, $t \in \mathcal{T}(\mathcal{F}, \mathcal{X})$ a linear term and σ a substitution $\sigma : \mathcal{X} \mapsto \mathcal{Q}$ such that $t\sigma \rightarrow_{\mathcal{A}}{}^* q$ (resp. $t\sigma \rightarrow_{\mathcal{A}}^{\not/*} q$). Then, there exists at least one substitution $\mu : \mathcal{X} \mapsto \mathcal{T}(\mathcal{F})$ such that $t\mu \rightarrow_{\mathcal{A}}{}^* q$ (resp. $t\mu \rightarrow_{\mathcal{A}}^{\not/*} q$). Furthermore, for all substitution μ such that $\mu = \{x \mapsto t' \mid x \in \mathcal{D}om(\sigma) \wedge t' \rightarrow_{\mathcal{A}}{}^* \sigma x$ (resp. $t' \rightarrow_{\mathcal{A}}^{\not/*} \sigma x$)$\}$, then $t\mu \rightarrow_{\mathcal{A}}{}^* q$ (resp. $t\mu \rightarrow_{\mathcal{A}}^{\not/*} q$).*

PROOF. We achieve the proof for $\rightarrow_{\mathcal{A}}^{\not/*}$ but it is similar for $\rightarrow_{\mathcal{A}}{}^*$. Let $t = C[x_1, \ldots, x_n]$. From $t\sigma \rightarrow_{\mathcal{A}}^{\not/*} q$ we know that $\{x_1, \ldots, x_n\} \subseteq \mathcal{D}om(\sigma)$. Let $q_1, \ldots, q_n \in \mathcal{A}$ be the states such that $\forall i = 1 \ldots n : q_i = x_i\sigma$. Since the automaton is filled, we know that there exists at least one term t_i per state q_i such that $t_i \rightarrow_{\mathcal{A}}^{\not/*} q_i$. Hence, there exists at least one substitution μ such that $\mu = \{x_1 \mapsto t_1, \ldots, x_n \mapsto t_n\}$. Let μ mapping x_i to any term t_i such that $t_i \rightarrow_{\mathcal{A}}^{\not/*} q_i$. We thus have $t\mu = C[x_1, \ldots, x_n]\mu = C[t_1, \ldots, t_n]$. Since $\forall i = 1 \ldots n : t_i \rightarrow_{\mathcal{A}}^{\not/*} q_i$, we have $C[t_1, \ldots, t_n] \rightarrow_{\mathcal{A}}^{\not/*} C[q_1, \ldots, q_n] = t\sigma$. But, by hypothesis, we know that $t\sigma \rightarrow_{\mathcal{A}}^{\not/*} q$. Hence, $t\mu \rightarrow_{\mathcal{A}}^{\not/*} t\sigma \rightarrow_{\mathcal{A}}^{\not/*} q$. $\square$

Theorem 97 *Let $\mathcal{A}, \mathcal{A}'$ be tree automata, $\mathcal{R}$ a TRS, E a set of equations and q_a, q_b states of $\mathcal{A}$ such that $\mathcal{A} \rightsquigarrow_E \mathcal{A}'$ and $\mathcal{A}' = \mathcal{A}\{q_a \mapsto q_b\}$. If $\mathcal{A}$ is $\mathcal{R}/E$-coherent then so is $\mathcal{A}'$. Furthermore, for all state q of $\mathcal{A}$ if a term s is a representative of q in $\mathcal{A}$ then it is a representative for $q\{q_a \mapsto q_b\}$ in $\mathcal{A}'$.*

PROOF. The proof of $\mathcal{R}/E$-coherence of $\mathcal{A}'$ has two parts, corresponding to the two conjuncts in Definition 93. Let $\mathcal{Q}$ and $\mathcal{Q}' = Q \setminus \{q_a\}$ be the set of states respectively of $\mathcal{A}$ and $\mathcal{A}'$.

1. We prove the first conjunct in Definition 93, that is, $\forall q \in \mathcal{Q}'.\ \exists s \in \mathcal{T}(\mathcal{F}).\ s \rightarrow_{\mathcal{A}}^{\not\varepsilon *} q \wedge \forall t \in \mathcal{T}(\mathcal{F}) : (t \rightarrow_{\mathcal{A}'}^{\not\varepsilon *} q \implies s =_E t)$. Let $q \in \mathcal{Q}'$. Then $q \in Q$, and, since $\mathcal{A}$ is $\mathcal{R}/E$-coherent, we know that

   (†) there exists a representative $s \in \mathcal{T}(\mathcal{F})$ such that $s \rightarrow_{\mathcal{A}}^{\not\varepsilon *} q$ and for all terms $t' \in \mathcal{T}(\mathcal{F}) : t' \rightarrow_{\mathcal{A}}^{\not\varepsilon *} q \implies s =_E t'$.

   We prove that the same term s can serve as representative of q in $\mathcal{A}'$.

   For this, we have to prove $s \rightarrow_{\mathcal{A}'}^{\not\varepsilon *} q$ and for all $t \in \mathcal{T}(\mathcal{F}) : t \rightarrow_{\mathcal{A}'}^{\not\varepsilon *} q \implies s =_E t$. The first statement, $s \rightarrow_{\mathcal{A}'}^{\not\varepsilon *} q$, is a consequence of $s \rightarrow_{\mathcal{A}}^{\not\varepsilon *} q$ and of Lemma 86. There remains to prove the second statement, that is for all $t \in \mathcal{T}(\mathcal{F}) : t \rightarrow_{\mathcal{A}'}^{\not\varepsilon *} q \implies s =_E t$, which is done by induction on t.

   - if height of t is 1, by definition of $\rightarrow_{\mathcal{A}'}^{\not\varepsilon *}$ we necessarily have $t \rightarrow q \in \mathcal{A}'$. Then by case on q:
     - if $q \neq q_b$ then $t \rightarrow q \in \mathcal{A}$ by definition of $\mathcal{A}'$. By letting $t' = t$ in (†) we obtain $s =_E t$, which concludes this case.
     - if $q = q_b$, then, by definition of $\mathcal{A}'$, either $t \rightarrow q_b \in \mathcal{A}$ or $t \rightarrow q_b \in \mathcal{A}$.
       * if $t \rightarrow q_b \in \mathcal{A}$ the proof is similar to that of the previous case.
       * if $t \rightarrow q_a \in \mathcal{A}$, then from $t \rightarrow_{\mathcal{A}}^{\not\varepsilon *} q_a$ and the fact that $\mathcal{A}$ is $\mathcal{R}/E$-coherent we know that there exists a representative u such that $u \rightarrow_{\mathcal{A}}^{\not\varepsilon *} q_a$. We now show that

         (‡) $u =_E s$, where s is the representative for $q = q_b$ from (†).

         Since $\mathcal{A} \rightsquigarrow_E \mathcal{A}'$ and $\mathcal{A}' = \{q_a \mapsto q_b\}$, we know that there exists an equation $l = r$ of E and a substitution $\sigma : \mathcal{X} \mapsto \mathcal{Q}$ such that $l\sigma \rightarrow_{\mathcal{A}}^{\not\varepsilon *} q_a$ and $r\sigma \rightarrow_{\mathcal{A}}^{\not\varepsilon *} q_b$. From this and Lemma 96, we get that there exists a substitution $\mu : \mathcal{X} \mapsto \mathcal{T}(\mathcal{F})$ such that $l\mu \rightarrow_{\mathcal{A}}^{\not\varepsilon *} q_a$ and $r\mu \rightarrow_{\mathcal{A}}^{\not\varepsilon *} q_b$. Since $l = r \in E$, we also have $l\mu =_E r\mu$. Then by using $\mathcal{R}/E$-coherence of $\mathcal{A}$ on state q_a we get that $u =_E l\mu$ and, similarly, on state q_b we get that $s =_E r\mu$. Finally, by transitivity of $=_E$ we obtain that $u =_E s$,which concludes the proof of (‡).
         From (‡) and $u =_E t$ (above) we get $s =_E t$, which concludes the proof in this case.
   - now, we assume that the property is true for any term t of height lesser of equal to n. Let us prove that the property is true for a term $t = f(t_1, \ldots, t_n) \rightarrow_{\mathcal{A}'}^{\not\varepsilon *} q$ whose height is lesser or equal to $n + 1$.

     By construction of tree automata derivation, from $t = f(t_1, \ldots, t_n) \rightarrow_{\mathcal{A}'}^{\not\varepsilon *} q$ we obtain that there exists states $q'_1, \ldots, q'_n$ such that: $f(t_1, \ldots, t_n) \rightarrow_{\mathcal{A}'}^{\not\varepsilon *} f(q'_1, \ldots, q'_n) \rightarrow_{\mathcal{A}'}^{\not\varepsilon *} q$. Like in the base case, we can discriminate between

several subcases where $q \neq q_b$, $q = q_b$ and $f(\ldots) \to q_b \in \mathcal{A}$, and $q = q_b$ and $f(\ldots) \to q_a \in \mathcal{A}$. For the last case, the proof that a representative of q_b is also a representative for terms recognized by q_a can be carried out like in the base case. So, let us assume that $q \neq q_b$ and focus on the other part of the proof where we assume that $\exists k \in \{1 \ldots n\}$ such that $q'_k = q_b$ and this state was obtained by the renaming of a q_a, i.e. that $f(q_1, \ldots, q_a, \ldots, q_n) \to q \in \mathcal{A}$. Note that we assume here that there is only one state q_k such that $q_k = q_a$ for simplicity, but the proof can be achieved in the same way with any number of states equal to q_a (or to q_b). Using those assumptions, we have: $f(t_1, \ldots, t_k, \ldots, t_n) \to^{\not{\epsilon}*}_{\mathcal{A}'} f(q_1, \ldots, q_b, \ldots, q_n) \to^{\not{\epsilon}*}_{\mathcal{A}'} q$ and $f(q_1, \ldots, q_a, \ldots, q_n) \to q \in \mathcal{A}$. Since $\mathcal{A}$ is $\mathcal{R}/E$-coherent we know that there exists representatives $s_1, \ldots, s_{k-1}, s_{k+1}, \ldots, s_n$ such that $\forall i \in \{1 \ldots n\}/\{k\} : s_i \to^{\not{\epsilon}*}_{\mathcal{A}} q_i$. Similarly, there exists a representative s_a for q_a such that $s_a \to^{\not{\epsilon}*}_{\mathcal{A}} q_a$. Let $u = f(s_1, \ldots, s_a, \ldots, s_n)$. We have $u = f(s_1, \ldots, s_a, \ldots, s_n) \to^{\not{\epsilon}*}_{\mathcal{A}} q$. Applying induction hypothesis on terms $\forall i \in \{1 \ldots n\}/\{k\} : t_i$, we obtain, furthermore that representatives $\forall i \in \{1 \ldots n\}/\{k\} : s_i$ of q_i in $\mathcal{A}$ are also representatives of q_i in $\mathcal{A}'$. Hence, $\forall i \in \{1 \ldots n\}/\{k\} : t_i \to^{\not{\epsilon}*}_{\mathcal{A}'} q_i \wedge t_i =_E s_i$. To conclude the proof, it is thus enough to prove that $s_a =_E t_k$. For this, we use the fact that, still by $\mathcal{R}/E$-coherence of $\mathcal{A}$, we know that there exists a representative s_b for q_b such that $s_b \to^{\not{\epsilon}*}_{\mathcal{A}} q_b$. On the other hand, applying induction hypothesis on $t_k \to^{\not{\epsilon}*}_{\mathcal{A}'} q_b$ we obtain that any representative of state q_b in $\mathcal{A}$ becomes a representative for q_b in $\mathcal{A}'$, and thus, $s_b =_E t_k$. Then we can apply the same reasoning as in the base case to prove that $s_a =_E s_b$. Since $\mathcal{A} \rightsquigarrow_E \mathcal{A}'$ and $\mathcal{A}' = \{q_a \mapsto q_b\}$, we know that there exists an equation $l = r$ of E and a substitution $\sigma : \mathcal{X} \mapsto \mathcal{Q}$ such that $l\sigma \to^{\not{\epsilon}*}_{\mathcal{A}} q_a$ and $r\sigma \to^{\not{\epsilon}*}_{\mathcal{A}} q_b$. From this result and Lemma 96, we get that there exists a substitution $\mu : \mathcal{X} \mapsto \mathcal{T}(\mathcal{F})$ such that $l\mu =_E r\mu$, $l\mu \to^{\not{\epsilon}*}_{\mathcal{A}} q_a$ and $r\mu \to^{\not{\epsilon}*}_{\mathcal{A}} q_b$. Then by using $\mathcal{R}/E$-coherence of $\mathcal{A}$ on state q_a we get that $s_a =_E l\mu$ and, similarly, on state q_b we get that $s_b =_E r\mu$. Finally, by transitivity of $=_E$ we obtain that $s_a =_E s_b$. Recall that, using the induction hypothesis, we have obtained that $\forall i \in \{1 \ldots n\}/\{k\} : s_i =_E t_i$. We also have that $s_a =_E s_b =_E t_k$. Thus $u = f(s_1, \ldots, s_a, \ldots, s_n) =_E f(s_1, \ldots, s_b, \ldots, s_n)$ and $f(s_1, \ldots, s_b, \ldots, s_n) =_E f(t_1, \ldots, t_k, \ldots, t_n) = t$, and by transitivity of $=_E$ we can conclude that $u =_E t$. Finally, by $\mathcal{R}/E$-coherence of $\mathcal{A}$ we get that $u =_E s$ and $s \to^{\not{\epsilon}*}_{\mathcal{A}} q$ where s is any representative of q in $\mathcal{A}$. From Lemma 86 and $s \to^{\not{\epsilon}*}_{\mathcal{A}} q$ we obtain that $s \to^{\not{\epsilon}*}_{\mathcal{A}'} q$. Using transitivity of $=_E$ on $s =_E u$ and $u =_E t$ we obtain that $s =_E t$. Hence, any representative s of q in $\mathcal{A}$ is still a representative of q in $\mathcal{A}'$.

2. In the second case of the proof, we have to prove that $\forall q \in \mathcal{Q}' : \exists s \in \mathcal{T}(\mathcal{F}) : s \to^{\not{\epsilon}*}_{\mathcal{A}'} q \wedge (\forall t \in \mathcal{T}(\mathcal{F}) : t \to^*_{\mathcal{A}'} q \implies s \to^*_{\mathcal{R}/E} t)$. We proceed by induction on the maximum height of t. Given a state $q \in \mathcal{Q}'$ to prove that it has (at least) all the representatives s of $\mathcal{A}$, we build a representative u of the same height than t, and then prove that $s =_E u$, $s \to^{\not{\epsilon}*}_{\mathcal{A}'} q$ and $u \to^*_{\mathcal{R}/E} t$, hence that $s \to^*_{\mathcal{R}/E} t$.

   - if height of t is 1, we have $t \to_{\mathcal{A}'} q_1 \to_{\mathcal{A}'} \ldots \to_{\mathcal{A}'} q_n$ where $q = q_n$. Then by

case on $q_1, \dots q_n$:

- $\forall i = 1 \dots n : q_i \neq q_b$, then we have a similar derivation in $\mathcal{A}$, i.e. $t \rightarrow_{\mathcal{A}} q_1 \rightarrow_{\mathcal{A}} \dots \rightarrow_{\mathcal{A}} q_n$ and by $\mathcal{R}/E$-coherence of $\mathcal{A}$ we obtain that there exists a representative s of q in $\mathcal{A}$, i.e. $s \rightarrow_{\mathcal{A}}^{\not\epsilon *} q$ and $s \rightarrow_{\mathcal{R}/E}^{*} t$. Lemma 86 implies that $s \rightarrow_{\mathcal{A}'}^{\not\epsilon *} q$. Hence any representative s of q in $\mathcal{A}$ becomes a representative of q in $\mathcal{A}'$.
- $\exists i = 1 \dots n : q_i = q_b$, i.e. $t \rightarrow_{\mathcal{A}'}^{*} q_i = q_b$ and $q_i \rightarrow_{\mathcal{A}'}^{*} q_n$. If $t \rightarrow_{\mathcal{A}}{}^{*} q_a$ (resp. $t \rightarrow_{\mathcal{A}}{}^{*} q_b$) and $q_a \rightarrow_{\mathcal{A}}{}^{*} q_n$ (resp. $q_b \rightarrow_{\mathcal{A}}{}^{*} q_n$) then the proof is similar to the previous case.
- $\exists i = 1 \dots n : q_i = q_b$, i.e. $t \rightarrow_{\mathcal{A}'}^{*} q_i = q_b$ and $q_i \rightarrow_{\mathcal{A}'}^{*} q_n$ and $t \rightarrow_{\mathcal{A}}{}^{*} q_a$ and $q_b \rightarrow_{\mathcal{A}}{}^{*} q_n$. Note that the proof is similar for the dual case where if $t \rightarrow_{\mathcal{A}}{}^{*} q_b$ and $q_a \rightarrow_{\mathcal{A}}{}^{*} q_n$. In the same way, even if there are some cycles in the chain $q_1, \dots, q_n$ and thus several occurrences of q_a and q_b, the proof is similar. So, let us assume that $t \rightarrow_{\mathcal{A}}{}^{*} q_a$ and $q_b \rightarrow_{\mathcal{A}}{}^{*} q_n$. From $\mathcal{R}/E$-coherence of $\mathcal{A}$ we obtain that there exist representatives s_a for q_a, s_b for q_b and s for q in $\mathcal{A}$ such that $s_a \rightarrow_{\mathcal{R}/E}^{*} t$ and $s \rightarrow_{\mathcal{R}/E}^{*} s_b$. Like in the base case of case (1), from the fact that $\mathcal{A} \rightsquigarrow_E \mathcal{A}'$ we can prove that there exists an equation $l = r$ and a substitution μ such that $l\mu =_E r\mu$, $l\mu \rightarrow_{\mathcal{A}}{}^{*} q_a$ and $r\mu \rightarrow_{\mathcal{A}}{}^{*} q_b$ and since $\mathcal{A}$ is coherent, $s_a =_E l\mu$ and $s_b =_E r\mu$. Hence $s_a =_E s_b$ and we thus have $s \rightarrow_{\mathcal{R}/E}^{*} s_b =_E s_a \rightarrow_{\mathcal{R}/E}^{*} t$, or more briefly $s \rightarrow_{\mathcal{R}/E}^{*} t$. Finally, by Lemma 86, we obtain that $s \rightarrow_{\mathcal{A}'}^{\not\epsilon *} q$ and thus s is a representative for q in $\mathcal{A}'$. Note that the case where $i = n$ can be treated in the same way and that in that case one has to show that s_a becomes also a representative for q in $\mathcal{A}'$ which is also the case since $s = s_b = s_a \rightarrow_{\mathcal{R}/E}^{*} t$.

- now, we assume that the property is true for any term t of height lesser of equal to n. Let us prove that the property is true for a term $t = f(t_1, \dots, t_n) \rightarrow_{\mathcal{A}'}^{*} q$ whose height is lesser or equal to $n+1$. By construction of tree automata derivation, from $t = f(t_1, \dots, t_n) \rightarrow_{\mathcal{A}'}^{*} q$ we obtain that there exists states $q'_1, \dots, q'_n$ such that: $f(t_1, \dots, t_n) \rightarrow_{\mathcal{A}'}^{*} f(q'_1, \dots, q'_n) \rightarrow_{\mathcal{A}'}^{*} q$. If $f(q'_1, \dots, q'_n) \rightarrow_{\mathcal{A}'} q_1 \rightarrow_{\mathcal{A}'} \dots \rightarrow_{\mathcal{A}'} q_m = q$ and some $q_i = q_b$, then we can apply the same technique as in the base case. Let us focus on the other part of the proof where $q \neq q_b$ and $\exists k \in \{1 \dots n\}$ such that $q'_k = q_b$ and this state was obtained by the renaming of a q_a, i.e. that $f(q_1, \dots, q_a, \dots, q_n) \rightarrow q \in \mathcal{A}$. As for case (1), we assume that there is only one q_a for sake of simplicity but the proof can be carried out for any number of q_a or q_b present in this transition. Using those assumptions, we have: $f(t_1, \dots, t_k, \dots, t_n) \rightarrow_{\mathcal{A}'}^{\not\epsilon *} f(q_1, \dots, q_b, \dots, q_n) \rightarrow_{\mathcal{A}'}^{\not\epsilon *} q$ and $f(q_1, \dots, q_a, \dots, q_n) \rightarrow q \in \mathcal{A}$. Since $\mathcal{A}$ is $\mathcal{R}/E$-coherent we know that there exists representatives $s_1, \dots, s_{k-1}, s_{k+1}, \dots, s_n$ such that $\forall i \in \{1 \dots n\}/\{k\}$: $s_i \rightarrow_{\mathcal{A}}^{\not\epsilon *} q_i$. Similarly, there exists a representative s_a for q_a such that $s_a \rightarrow_{\mathcal{A}}^{\not\epsilon *} q_a$. Let $u = f(s_1, \dots, s_a, \dots, s_n)$. We have $u = f(s_1, \dots, s_a, \dots, s_n) \rightarrow_{\mathcal{A}}^{\not\epsilon *}$

q. Applying induction hypothesis on terms $\forall i \in \{1 \dots n\}/\{k\} : t_i$, we obtain, furthermore that representatives $\forall i \in \{1 \dots n\}/\{k\} : s_i$ of q_i in $\mathcal{A}$ are also representatives of q_i in $\mathcal{A}'$. Hence, $\forall i \in \{1 \dots n\}/\{k\} : t_i \rightarrow^{\not e *}_{\mathcal{A}} q_i \wedge s_i \rightarrow^*_{\mathcal{R}/E} t_i$. To conclude the proof, it is thus enough to prove that $s_a \rightarrow^*_{\mathcal{R}/E} t_k$. For this, we use the fact that, still by $\mathcal{R}/E$-coherence of $\mathcal{A}$, we know that there exists a representative s_b for q_b such that $s_b \rightarrow^{\not e *}_{\mathcal{A}} q_b$. On the other hand, applying induction hypothesis on $t_k \rightarrow^{\not e *}_{\mathcal{A}'} q_b$ we obtain that any representative of state q_b in $\mathcal{A}$ becomes a representative for q_b in $\mathcal{A}'$, and thus, $s_b \rightarrow^*_{\mathcal{R}/E} t_k$. Then we can apply the same reasoning as in the base case to prove that $s_a =_E s_b$. Since $\mathcal{A} \rightsquigarrow_E \mathcal{A}'$ and $\mathcal{A}' = \{q_a \mapsto q_b\}$, we know that there exists an equation $l = r$ of E and a substitution $\mu : \mathcal{X} \mapsto \mathcal{T}(\mathcal{F})$ such that $l\mu =_E r\mu$, $l\mu \rightarrow^{\not e *}_{\mathcal{A}} q_a$ and $r\mu \rightarrow^{\not e *}_{\mathcal{A}} q_b$. Then by using $\mathcal{R}/E$-coherence of $\mathcal{A}$ on state q_a we get that $s_a =_E l\mu$ and, similarly, on state q_b we get that $s_b =_E r\mu$. Finally, by transitivity of $=_E$ we obtain that $s_a =_E s_b$. Then, from $s_a =_E s_b$ and $s_a =_E s_b \rightarrow^*_{\mathcal{R}/E} t_k$ we obtain that $s_a \rightarrow^*_{\mathcal{R}/E} t_k$. Besides of this, recall that using the induction hypothesis, we have obtained that $\forall i \in \{1 \dots n\}/\{k\} : s_i \rightarrow^*_{\mathcal{R}/E} t_i$. Thus $u = f(s_1, \dots, s_a, \dots, s_n) \rightarrow^*_{\mathcal{R}/E} f(t_1, \dots, t_k, \dots, t_n) = t$. Finally, by $\mathcal{R}/E$-coherence of $\mathcal{A}$ we get that $u =_E s$ and $s \rightarrow^{\not e *}_{\mathcal{A}} q$ where s is any representative of q in $\mathcal{A}$. From Lemma 86 and $s \rightarrow^{\not e *}_{\mathcal{A}} q$ we obtain that $s \rightarrow^{\not e *}_{\mathcal{A}'} q$. Then, from $s =_E u$ and $u \rightarrow^*_{\mathcal{R}/E} t$ we obtain that $s \rightarrow^*_{\mathcal{R}/E} t$. Hence, any representative s of q in $\mathcal{A}$ is still a representative of q in $\mathcal{A}'$ and can be rewritten by $\rightarrow_{\mathcal{R}/E}$ into t.

□

3.2.3 One step of equational completion

Given a tree automaton $\mathcal{A}$ and a TRS $\mathcal{R}$, the tree automata completion algorithm, proposed in (Genet, 1998; Feuillade et al., 2004), computes a tree automaton $\mathcal{A}^k_{\mathcal{R}}$ such that $\mathcal{L}(\mathcal{A}^k_{\mathcal{R}}) = \mathcal{R}^*(\mathcal{L}(\mathcal{A}))$ when it is possible (for the classes of TRSs where an exact computation is possible) and such that $\mathcal{L}(\mathcal{A}^k_{\mathcal{R}}) \supseteq \mathcal{R}^*(\mathcal{L}(\mathcal{A}))$ otherwise.

Recall that completion builds a sequence $\mathcal{A}^0_{\mathcal{R}}, \mathcal{A}^1_{\mathcal{R}}, \dots, \mathcal{A}^k_{\mathcal{R}}$ of automata such that if $s \in \mathcal{L}(\mathcal{A}^i_{\mathcal{R}})$ and $s \rightarrow_{\mathcal{R}} t$ then $t \in \mathcal{L}(\mathcal{A}^{i+1}_{\mathcal{R}})$. As in the standard completion, we search for critical pairs $l\sigma \rightarrow^*_{\mathcal{A}^i_{\mathcal{R}}} q$ and $r\sigma \not\rightarrow^*_{\mathcal{A}^i_{\mathcal{R}}} q$ with $l \rightarrow r \in \mathcal{R}$ and $q \in \mathcal{A}^i_{\mathcal{R}}$. In equational completion, critical pairs are solved as follows. For every critical pair detected between $\mathcal{R}$ and $\mathcal{A}^i_{\mathcal{R}}$ such that $r\sigma \not\rightarrow^*_{\mathcal{A}^i_{\mathcal{R}}} q$, the tree automaton $\mathcal{A}^{i+1}_{\mathcal{R}}$ is constructed by adding new transitions $r\sigma \rightarrow q'$ and $q' \rightarrow q$ to $\mathcal{A}^i_{\mathcal{R}}$ such that $\mathcal{A}^{i+1}_{\mathcal{R}}$ recognizes $r\sigma$ in q, i.e. $r\sigma \rightarrow^*_{\mathcal{A}^{i+1}_{\mathcal{R}}} q$.

$$\begin{tikzcd}
l\sigma \arrow[r, "\mathcal{R}"'] \arrow[d, "\mathcal{A}^i_{\mathcal{R}}"'] & r\sigma \arrow[d, "\mathcal{A}^{i+1}_{\mathcal{R}}"] \\
q & q' \arrow[l, "\mathcal{A}^{i+1}_{\mathcal{R}}"]
\end{tikzcd}$$

As before, the transition $r\sigma \to q'$ is not necessarily a normalized transition of the form $f(q_1, \ldots, q_n) \to q'$ and so it has to be normalized first. One of the aims of this paper is to replace normalization rules by equations. Using equations require to build $\mathcal{R}/E$-coherent tree automata. This is the reason why, in order to have precise approximations, the resolution of critical pairs is not identical to the one defined in (Genet, 1998). Indeed, in (Genet, 1998), a critical pair was solved in the following way:

$$\begin{tikzcd} l\sigma \arrow[r, "\mathcal{R}"'] \arrow[d, "\mathcal{A}^i_{\mathcal{R}}"', "*"] & r\sigma \arrow[dl, bend left, "*"] \\ q & \mathcal{A}^{i+1}_{\mathcal{R}} \end{tikzcd}$$

The transition $r\sigma \to q$ was added directly to $\mathcal{A}^{i+1}_{\mathcal{R}}$. In the version we propose here, we add transitions so that $r\sigma \to q'$ and an epsilon transition $q' \to q$. This is necessary to preserve the precision of approximations achieved using the simplification relation defined above. This is explained in the example below.

Example 98 *Let $\mathcal{A}$ be a tree automaton, q_1, q_2, q_3, q_4 states of $\mathcal{A}$, $\mathcal{R}$ a TRS, $E = \{s = u\}$ a set of equations, s, t, u, v be terms such that $s \to_{\mathcal{R}} t$ and $u \to_{\mathcal{R}} v$. Assuming that $\mathcal{A}$ is a tree automaton produced by the completion algorithm of (Genet, 1998), if $s \to_{\mathcal{A}}{}^* q_1$ and $u \to_{\mathcal{A}}{}^* q_2$ then we have the following situation:*

$$\begin{tikzcd} s \arrow[r, "\mathcal{R}"'] \arrow[d, "\mathcal{A},\not{\epsilon}"', "*"] & t \arrow[d, "*"', "\mathcal{A},\not{\epsilon}"] \\ q_1 & q_1 \end{tikzcd} \qquad \begin{tikzcd} u \arrow[r, "\mathcal{R}"'] \arrow[d, "\mathcal{A},\not{\epsilon}"', "*"] & v \arrow[d, "*"', "\mathcal{A},\not{\epsilon}"] \\ q_2 & q_2 \end{tikzcd}$$

Now recall that $E = \{s = u\}$, if we use the simplification relation defined above, $A \rightsquigarrow_E \mathcal{A}'$ such that:

$$\begin{tikzcd} s \arrow[r, "\mathcal{R}"'] \arrow[d, "\mathcal{A}',\not{\epsilon}"', "*"] & t \arrow[d, "*"', "\mathcal{A}',\not{\epsilon}"] \\ q_1 & q_1 \end{tikzcd} \qquad \begin{tikzcd} u \arrow[r, "\mathcal{R}"'] \arrow[d, "\mathcal{A}',\not{\epsilon}"', "*"] & v \arrow[d, "*"', "\mathcal{A}',\not{\epsilon}"] \\ q_1 & q_1 \end{tikzcd}$$

In other words, s and u are equivalent in $\mathcal{A}'$ (they are recognized by q_1). This is coherent with E. However, t and v are also equivalent to s and u. If using completion for verification purposes this approximation may be too strong. If a program evolves from a state s to t or from u to v ($\mathcal{R}$ encode the transition function of the program) and if the approximation merges together states s and u, from

$$\begin{tikzcd} s \arrow[r, "\mathcal{R}"'] \arrow[d, equal, "E"'] & t \\ u \arrow[r, "\mathcal{R}"'] & v \end{tikzcd} \quad \text{a representation of the form} \quad \begin{tikzcd} & t \\ s, u \arrow[ur, "\mathcal{R}"] \arrow[dr, "\mathcal{R}"'] & \\ & v \end{tikzcd} \quad \text{is more precise than} \quad s, u \xrightarrow[\mathcal{R}]{} t, v$$

The proposed refinement of the completion procedure produces:

$$\begin{tikzcd} s \arrow[r, "\mathcal{R}"'] \arrow[d, "\mathcal{A},\not\epsilon"', "*"] & t \arrow[d, "*", "\mathcal{A},\not\epsilon"] \\ q_1 & q_3 \arrow[l, "\epsilon"] \end{tikzcd} \qquad \begin{tikzcd} u \arrow[r, "\mathcal{R}"'] \arrow[d, "\mathcal{A},\not\epsilon"', "*"] & v \arrow[d, "*", "\mathcal{A},\not\epsilon"] \\ q_2 & q_4 \arrow[l, "\epsilon"] \end{tikzcd}$$

and the approximation equation of E will simplify this automaton into:

$$\begin{tikzcd} s \arrow[r, "\mathcal{R}"'] \arrow[d, "\mathcal{A},\not\epsilon"', "*"] & t \arrow[d, "*", "\mathcal{A},\not\epsilon"] \\ q_1 & q_3 \arrow[l, "\epsilon"] \end{tikzcd} \qquad \begin{tikzcd} u \arrow[r, "\mathcal{R}"'] \arrow[d, "\mathcal{A},\not\epsilon"', "*"] & v \arrow[d, "*", "\mathcal{A},\not\epsilon"] \\ q_1 & q_4 \arrow[l, "\epsilon"] \end{tikzcd}$$

where s and u are recognized by the same state and t and v are recognized by distinct states.

In this new completion algorithm, since approximations are performed by simplification of the tree automaton as defined in Section 3.2.1, the definition of normalization can be simplified. In particular, it does no longer need to be parameterized by an abstraction function α. The simplified normalization function normalize subterms by either states of $\mathcal{Q}$ (using transitions of Δ) or new states. A state q of $\mathcal{Q}$ is used to normalize a term t if $t \rightarrow_{\Delta}^{*} q$. Normalizing by reusing states of $\mathcal{Q}$ and transitions of Δ permits to preserve the determinism of $\rightarrow_{\Delta}^{\not\epsilon}$. Indeed, $\rightarrow_{\Delta}^{\not\epsilon}$ can be kept deterministic during completion though $\rightarrow_{\Delta}$ cannot.

Definition 99 (Normalization) *Let Λ be the set of all transitions defined on $\mathcal{T}(\mathcal{F} \cup \mathcal{Q})$. Let $s \in \mathcal{T}(\mathcal{F} \cup \mathcal{Q})$ and $\Delta \subseteq \Lambda$ be a set of transitions. A* new state *is a state of $\mathcal{Q}$ not occurring in Δ. The function $Norm_{\Delta}$ has type $\mathcal{T}(\mathcal{F} \cup \mathcal{Q}) \mapsto (\mathcal{Q} \times \mathcal{P}(\Lambda))$. $Norm_{\Delta}(s)$ is inductively defined by:*

1. *if $s \in \mathcal{Q}$ then $Norm_{\Delta}(s) = (s, \emptyset)$,*
2. *if $s = f(t_1, \ldots, t_n)$, then $Norm_{\Delta}(s) = (q, \{f(q'_1, \ldots, q'_n) \rightarrow q\} \cup \bigcup_{i=1}^{n} N_i)$ where for $i = 1 \ldots n$, we choose (q'_i, N_i) and q such that:*
   - $(q'_i, N_i) = \begin{cases} (q_i, \emptyset) \text{ if } q_i \in \{u \mid t \rightarrow_{\Delta}^{\not\epsilon *} u\} \neq \emptyset \\ Norm_{\Delta}(t_i), \text{ otherwise} \end{cases}$
   - *q is either*
     - *the right-hand side of a transition $f(q'_1, \ldots, q'_n) \rightarrow q$ if such a transition exists in Δ*
     - *or a new state, otherwise.*

$\diamond$

Note that using this definition of $Norm_\Delta$ it is possible to normalize with new states as well as to reuse the states of $\mathcal{Q}$ and transitions of Δ. Thus it is either possible to preserve the determinism of $\rightarrow_\Delta^{\not\varepsilon}$ or not, depending on the objective of the approximation. Now, we can use this definition to formally define one step of completion. A step of completion only concerns the resolution of critical pairs and not how approximations are performed. The approximation step has been defined in Section 3.2.1 and the full completion algorithm will be defined in Section 3.2.4. We first restrict the set of critical pairs to the non trivial ones. A critical pair is trivial if one of the states substituted to the variables is rewritten by an epsilon transition.

Definition 100 (Trivial critical pair) *Let $l \rightarrow r$ be a rule of $\mathcal{R}$ such that $l = C[x_1, \ldots x_n]$, $\mathcal{A} = \langle \mathcal{F}, \mathcal{Q}, \mathcal{Q}_f, \Delta \rangle$ a tree automaton, $q, q_1, \ldots, q_n \in \mathcal{Q}$ and $\sigma = \{x_1 \mapsto q_1, \ldots, x_n \mapsto q_n\}$. The critical pair*

$$\begin{array}{ccc} l\sigma & \xrightarrow[\mathcal{R}]{} & r\sigma \\ {\scriptstyle \mathcal{A}}\downarrow{\scriptstyle *} & & \\ q & & \end{array}$$

is trivial if all derivations $l\sigma = C[q_1, \ldots, q_n] \rightarrow_{\mathcal{A}}{}^* q$ contains at least one epsilon step between q_i $(1 \leq i \leq n)$ and $q' \in \mathcal{Q}$, i.e. the derivation is of the form $l\sigma = C[q_1, \ldots, q_i, \ldots, q_n] \rightarrow_{\mathcal{A}} C[q_1, \ldots, q', \ldots, q_n] \rightarrow_{\mathcal{A}}{}^* q$. ◇

It is not necessary to consider critical pairs during completion. Here is an example showing why.

Example 101 *Let $\mathcal{R} = \{f(x) \rightarrow g(x)\}$ and $\mathcal{A}$ the tree automaton whose transition set Δ is $\{f(q_0) \rightarrow q_f, a \rightarrow q_0, b \rightarrow q_1, q_1 \rightarrow q_0\}$. Let $\sigma_1 = \{x \mapsto q_0\}$ and $\sigma_2 = \{x \mapsto q_1\}$. The critical pair $l\sigma_1 \rightarrow_{\mathcal{A}} q_f$ is non trivial but $l\sigma_2 \rightarrow_{\mathcal{A}} q_f$ is, since $f(q_1) \rightarrow_{\mathcal{A}} f(q_0) \rightarrow_{\mathcal{A}} q_f$. Applying completion on the first critical pair produces the new transitions $g(q_0) \rightarrow q_2, q_2 \rightarrow q_f$. Applying completion on the second critical pair gives the new transitions $g(q_1) \rightarrow q_3, q_3 \rightarrow q_f$. However, note that this second set of transitions is useless to add because with the first set and $\mathcal{A}$ we already have $g(q_1) \rightarrow g(q_0) \rightarrow\rightarrow q_2 \rightarrow q_f$.*

Definition 102 (Set of (non trivial) critical pairs) *Let a TRS $\mathcal{R}$ and a tree automaton $\mathcal{A} = aut$. The set of (non trivial) critical pairs between $\mathcal{R}$ and $\mathcal{A}$ is $CP(\mathcal{R}, \mathcal{A}) = \{(l \rightarrow r, \sigma, q) \mid l \rightarrow r \in \mathcal{R}$ and $q \in \mathcal{Q}$ and $\sigma \in \Sigma(\mathcal{Q}, \mathcal{X})$ and $l\sigma \rightarrow_{\mathcal{A}}^* q$ and $r\sigma \not\rightarrow_{\mathcal{A}}^* q$ and $(l \rightarrow r, \sigma, q)$ non trivial$\}$.* ◇

Definition 103 (One step automaton completion) *Let $\mathcal{A} = \langle \mathcal{F}, \mathcal{Q}, \mathcal{Q}_f, \Delta \rangle$ be a tree automaton, $\mathcal{R}$ be a left-linear TRS. The one step completed automaton is $\mathcal{C}_\mathcal{R}(\mathcal{A}) = \langle \mathcal{F}, \mathcal{Q}', \mathcal{Q}_f, \Delta' \rangle$ where:*

$$\Delta' = \Delta \cup \bigcup_{(l \rightarrow r, q, \sigma) \in CP(\mathcal{R}, \mathcal{A})} NewT \cup \{q' \rightarrow q\}$$

where $(q', NewT) = Norm_\Delta(r\sigma)$ and $\mathcal{Q}'$ contains all the states of Δ'. ◇

Example 104 *Let $\mathcal{A}$ be a tree automaton with $\Delta = \{f(q_1) \to q_0, a \to q_1, g(q_1) \to q_2\}$.*

- *If $\mathcal{R} = \{f(x) \to x\}$ then there is a critical pair $f(x)\sigma \to_{\mathcal{A}}{}^* q_0$ and $f(x)\sigma \to_{\mathcal{R}} x\sigma$ where $\sigma = \{x \mapsto q_1\}$. Since $x\sigma = q_1$, and $Norm_\Delta(q_1) = (q_1, \emptyset)$, we obtain that $\Delta' = \Delta \cup \{q_1 \to q_0\}$;*
- *If $\mathcal{R} = \{f(a) \to g(a)\}$ then there is a critical pair $f(a)\sigma \to_{\mathcal{A}}{}^* q_0$ and $f(a)\sigma \to_{\mathcal{R}} g(a)\sigma$ with $\sigma = \emptyset$. Since $Norm_\Delta(g(a)) = (q_2, \emptyset)$, we obtain that $\Delta' = \Delta \cup \{q_2 \to q_0\}$;*
- *If $\mathcal{R} = \{f(x) \to f(g(x))\}$ then the critical pair is $f(x)\sigma \to_{\mathcal{A}}{}^* q_0$ and $f(x)\sigma \to_{\mathcal{R}} f(g(x))\sigma$ with $\sigma = \{x \mapsto q_1\}$. Since $Norm_\Delta(f(g(q_1))) = (q_3, \{f(q_2) \to q_3\})$, we obtain that $\Delta' = \Delta \cup \{f(q_2) \to q_3, q_3 \to q_0\}$.*

Now, we show that completion steps preserve $\mathcal{R}/E$-coherence. This is done in two steps. Assume that completion finds a term $l\sigma \to_{\mathcal{A}}{}^* q'$ such that $l\sigma \to_{\mathcal{R}} r\sigma$. In a first lemma, we first show that adding a term $t = r\sigma$ alone preserves $\mathcal{R}/E$-coherence. Then, in a second lemma, we show that if we perform a complete $\mathcal{C}_\mathcal{R}$ step this also preserve $\mathcal{R}/E$-coherence. With regards to the first lemma, this essentially consists in proving that adding an epsilon-transition $q \to q'$, such that $l\sigma \to_{\mathcal{A}}{}^* q$ and $r\sigma \to_{\mathcal{A}}{}^* q'$, to the tree automaton preserves $\mathcal{R}/E$-coherence.

Lemma 105 *Let $\mathcal{A} = \langle \mathcal{F}, \mathcal{Q}, \mathcal{Q}_f, \Delta \rangle$ be a tree automaton, $\mathcal{R}$ a TRS, E a set of equations, $t \in \mathcal{T}(\mathcal{F} \cup \mathcal{Q})$ and $(q', NewT) = Norm_\Delta(t)$. If $\mathcal{A}$ is $\mathcal{R}/E$-coherent, then so is the automaton $\mathcal{A}' = \langle \mathcal{F}, \mathcal{Q}', \mathcal{Q}_f, \Delta \cup NewT \rangle$.*

PROOF. We prove this by induction on height of t.

- if t is of height 0 then $t = q \in \mathcal{Q}$ and $NewT = \emptyset$. Hence since $\mathcal{A}' = \mathcal{A}$, $\mathcal{A}'$ is trivially $\mathcal{R}/E$-coherent.
- if t is of height 1 then $NewT = \{t \to q'\}$. If $t \to q' \in \Delta$ then $\mathcal{A}' = \mathcal{A}$ and $\mathcal{A}'$ trivially $\mathcal{R}/E$-coherent. Otherwise, if $t \to q' \notin \Delta$ then q' is necessarily a new state, by Definition 99 and t is the only term recognized by q' in $\mathcal{A}'$. Hence, t is also the representative of state q' in $\mathcal{A}'$, and $\mathcal{R}/E$-coherence of q' and thus of $\mathcal{A}'$ is trivial.
- let us assume that the property holds for terms of height lesser or equal to n. We now prove that the property holds for terms of height lesser or equal to $n+1$. Let $t = f(t_1, \ldots, t_n)$. By definition of normalization, $Norm(f(t_1, \ldots, t_n)) = (q', NewT)$ where $NewT = \{f(q'_1, \ldots, q'_n) \to q'\} \cup \bigcup_{i=1}^n N_i)$. For N_i either it is empty and $q'_i = q_i$ if there exist a state q_i such that $t_i \to_\Delta^{\not\epsilon *} q_i$, or $N_i = Norm_\Delta(t_i)$. In both cases the automaton $\langle \mathcal{F}, \mathcal{Q}, \mathcal{Q}_f, \Delta \cup N_i \rangle$ is $\mathcal{R}/E$-coherent. This is trivially the case if $N_i = \emptyset$ and it can be shown using the induction hypothesis on $Norm_\Delta(t_i)$ otherwise. We can also successively use this result and build the tree automaton $\mathcal{A}'' = \langle \mathcal{F}, \mathcal{Q}, \mathcal{Q}_f, \Delta \cup \bigcup_{i=1}^n N_i \rangle$ which is thus $\mathcal{R}/E$-coherent. The last thing to prove is that adding $f(q'_1, \ldots, q'_n) \to q'$ preserves coherence of q'. If $f(q'_1, \ldots, q'_n) \to q' \in \Delta$ then this is trivial since $\mathcal{A}' = \mathcal{A}''$ which is coherent. Otherwise, q' is a new state

and we have to prove its $\mathcal{R}/E$-coherence. Since $f(q'_1, \ldots, q'_n) \to q'$, any term u such that $u \to^*_{\mathcal{A}'} q'$ is necessarily of the form $f(u_1, \ldots, u_n)$ and $f(u_1, \ldots, u_n) \to^*_{\mathcal{A}'} f(q'_1, \ldots, q'_n) \to_{\mathcal{A}'} q'$. Since the tree automaton $\mathcal{A}'$ is coherent for $q'_1, \ldots, q'_n$ we know that there exists representatives $\forall i = 1 \ldots n : s_i \to^{\not{\epsilon}*}_{\mathcal{A}'} q_i \wedge s_i \to^*_{\mathcal{R}/E} u_i$. Hence, $s = f(s_1, \ldots, s_n) \to^{\not{\epsilon}*}_{\mathcal{A}'} q'$ and $f(s_1, \ldots, s_n) \to^*_{\mathcal{R}/E} f(u_1, \ldots, u_n)$, thus s is a representative for q', and q' is $\mathcal{R}/E$-coherent in $\mathcal{A}'$.

□

Lemma 106 *Let $\mathcal{A}$ be a tree automaton, $\mathcal{R}$ a TRS, E a set of equations. If $\mathcal{A}$ is $\mathcal{R}/E$-coherent then so is $\mathcal{C}_{\mathcal{R}}(\mathcal{A})$.*

PROOF. Let $\mathcal{A} = \langle \mathcal{F}, \mathcal{Q}, \mathcal{Q}_f, \Delta \rangle$. We prove that for any rewrite rule $l \to r$, for any substitution $\sigma : \mathcal{X} \mapsto \mathcal{Q}$ and for any state $q \in \mathcal{Q}$ such that $l\sigma \to_{\mathcal{A}}{}^* q$, if $(q', NewT) = Norm_{\Delta}(r\sigma)$ then adding transitions $q \to q'$ and $NewT$ to $\mathcal{A}$ results into a tree automaton $\mathcal{A}'$ that is $\mathcal{R}/E$-coherent. Thanks to Lemma 105, we know that adding $NewT$ to $\mathcal{A}$ produces a $\mathcal{R}/E$-coherent tree automaton, hence in particular $\exists s' \in \mathcal{T}(\mathcal{F}) : s' \to^{\not{\epsilon}*}_{\mathcal{A}'} q' \wedge [\forall t \in \mathcal{T}(\mathcal{F}) : (t \to^*_{\mathcal{A}'} q' \implies s' \to^*_{\mathcal{R}/E} t)]$. Now it is enough to prove that adding $q' \to q$ preserves $\mathcal{R}/E$-coherence of state q. For all term $t \in \mathcal{T}(\mathcal{F})$ such that $t \to^{\not{\epsilon}*}_{\mathcal{A}'} q' \to q$, we know that $t =_E s'$. Furthermore, let $\mu' : X \mapsto \mathcal{T}(\mathcal{F})$ such that for all variable x of σ $\mu'(x) = s_x$ where s_x is a representative of state $\sigma(x)$, i.e. $s_x \to^{\not{\epsilon}*}_{\mathcal{A}'} \sigma(x)$. From $r\sigma \to^{\not{\epsilon}*}_{\mathcal{A}'} q'$, the fact that μ maps variables of σ to terms $s_x \to^{\not{\epsilon}*}_{\mathcal{A}'} \sigma(x)$, and Lemma 96, we obtain that $r\mu' \to^{\not{\epsilon}*}_{\mathcal{A}'} q'$. Hence, by $\mathcal{R}/E$-coherence of q' in $\mathcal{A}'$, we have that $r\mu' =_E s'$. Since $\mathcal{A}$ is coherent, for state q we have that $\exists s \in \mathcal{T}(\mathcal{F}) : s \to^{\not{\epsilon}*}_{\mathcal{A}} q \wedge [\forall t \in \mathcal{T}(\mathcal{F}) : (t \to_{\mathcal{A}}{}^* q \implies s \to^*_{\mathcal{R}/E} t)]$. We can instantiate this lemma for $l\mu' \to^{\not{\epsilon}*}_{\mathcal{A}'} q$, hence $s \to^*_{\mathcal{R}/E} l\mu'$. Finally, we have $s \to^*_{\mathcal{R}/E} l\mu' \to r\mu' =_E s' \to^*_{\mathcal{R}/E} t$, or $s \to^*_{\mathcal{R}/E} t$ for short. Thus s is a representative for t recognized by state q in $\mathcal{A}'$. □

3.2.4 The full Completion Algorithm

We here define the full equational completion algorithm. This section also contains the proof that completion produces a tree automaton that recognizes an over-approximation of reachable terms (Completeness). Finally, we give the proof that this automaton is $\mathcal{R}/E$-coherent (Correctness w.r.t. $\mathcal{R}/E$ reachable terms).

Definition 107 (Automaton completion) *Let $\mathcal{A}$ be a tree automaton, $\mathcal{R}$ a TRS and E a set of equations.*

- $\mathcal{A}^0_{\mathcal{R},E} = \mathcal{A}$,
- $\mathcal{A}^{n+1}_{\mathcal{R},E} = \mathcal{A}'$ *where* $\mathcal{C}_{\mathcal{R}}(\mathcal{A}^n_{\mathcal{R},E}) \rightsquigarrow^!_E \mathcal{A}'$, *and* μ_{n+1} *is the renaming such that* $\mathcal{A}' = \mathcal{C}_R(\mathcal{A}^n_{\mathcal{R},E})\mu_{n+1}$
- $\mathcal{A}^*_{\mathcal{R},E}$ *is a fixpoint* $\mathcal{A}^*_{\mathcal{R},E} = \mathcal{A}^k_{\mathcal{R},E} = \mathcal{A}^{k+1}_{\mathcal{R},E}$ *with* $k \in \mathbb{N}$.

◇

Note that, like for the standard completion, $\mathcal{A}^*_{\mathcal{R},E}$ does not exists in general but it can be computed provided that E ensures the termination.

Example 108 *Let $\mathcal{R} = \{f(x,y) \to f(s(x),s(y))\}$, $E = \{s(s(x)) = s(x)\}$ and $\mathcal{A}^0$ be the tree automaton with set of transitions $\Delta = \{f(q_a,q_b) \to q_0, a \to q_a, b \to q_b\}$, i.e. $\mathcal{L}(\mathcal{A}^0) = \{f(a,b)\}$. The completion ends after two completion steps on $\mathcal{A}^2_{\mathcal{R},E}$ which is a fixpoint. Completion steps are summed up in the following table. To simplify the presentation, we do not repeat the common transitions, i.e. $\mathcal{A}^i_{\mathcal{R},E}$ is supposed to contain all transitions of $\mathcal{A}^0,\ldots,\mathcal{A}^{i-1}_{\mathcal{R},E}$.*

$\mathcal{A}^0$	$\mathcal{A}^1_{\mathcal{R},E}$	$\mathcal{C}_{\mathcal{R}}(\mathcal{A}^1_{\mathcal{R},E})$	$\mathcal{A}^2_{\mathcal{R},E}$
$f(q_a,q_b) \to q_0$ $a \to q_a$ $b \to q_b$	$f(q_1,q_2) \to q_3$ $s(q_a) \to q_1$ $s(q_b) \to q_2$ $q_3 \to q_0$	$f(q_4,q_5) \to q_6$ $s(q_1) \to q_4$ $s(q_2) \to q_5$ $q_6 \to q_3$	$f(q_1,q_2) \to q_6$ $s(q_1) \to q_1$ $s(q_2) \to q_2$
$\mathcal{L}(\mathcal{A}^0) = \{f(a,b)\}$	$\mathcal{L}(\mathcal{A}^1_{\mathcal{R},E}) = \{f(a,b),$ $f(s(a),s(b))\}$	$\mathcal{L}(\mathcal{C}_{\mathcal{R}}(\mathcal{A}^1_{\mathcal{R},E})) = \{f(a,b),$ $f(s(a),s(b)),$ $f(s(s(a)),s(s(b))\}$	$\mathcal{L}(\mathcal{A}^2_{\mathcal{R},E}) =$ $\{f(s^*(a),s^*(b))\}$

The automaton $\mathcal{A}^1_{\mathcal{R},E}$ is exactly $\mathcal{C}_{\mathcal{R}}(\mathcal{A}^0)$ since equations do not apply. Then $\mathcal{C}_{\mathcal{R}}(\mathcal{A}^1_{\mathcal{R},E})$ contains all the transitions of $\mathcal{A}^1_{\mathcal{R},E}$ plus those obtained by the resolution of the critical pair $f(q_1,q_2) \to_{\mathcal{A}^*} q_3$ and $f(q_1,q_2) \to_{\mathcal{R}} f(s(q_1),s(q_2))$. Solving this critical pair according, adds the transitions shown in the above table. However, on this last automaton, simplification can be applied as follows:

$$\begin{tikzcd} s(s(q_a)) \arrow[r, equal, "E"'] \arrow[d, "\mathcal{A},\not\epsilon"', "*"] & s(q_a) \arrow[d, "*"', "\mathcal{A},\not\epsilon"] \\ q_4 & q_1 \end{tikzcd} \qquad \begin{tikzcd} s(s(q_b)) \arrow[r, equal, "E"'] \arrow[d, "\mathcal{A},\not\epsilon"', "*"] & s(q_b) \arrow[d, "*"', "\mathcal{A},\not\epsilon"] \\ q_5 & q_2 \end{tikzcd}$$

Hence, $\mathcal{A}^2_{\mathcal{R},E}$ is obtained from $\mathcal{C}_{\mathcal{R}}(\mathcal{A}^1_{\mathcal{R},E})$ by renaming q_4 by q_1 and q_5 by q_2, i.e. $\mathcal{A}^2_{\mathcal{R},E} = \mathcal{C}_{\mathcal{R}}(\mathcal{A}^1_{\mathcal{R},E})\{q_4 \mapsto q_1, q_5 \mapsto q_2\}$.

Some other practical examples will be given in Chapter 4. Now, we can state the completeness and correctness theorems. We show that tree automata completion is complete w.r.t. $\mathcal{R}^*$ and correct w.r.t. $\mathcal{R}^*_E$. The completeness theorem ensures that the completed automaton $\mathcal{A}^*_{\mathcal{R},E}$ recognizes all $\mathcal{R}$-reachable terms (but not all $\mathcal{R}/E$-reachable terms). The correctness theorem guarantees that all terms recognized by $\mathcal{A}^*_{\mathcal{R},E}$ are only $\mathcal{R}/E$-reachable terms. This distinction is important to use this technique for verification. Indeed, if $\mathcal{R}$ models the program and E defines the approximation then it is natural to focus the theorem on the over-approximation of $\mathcal{R}$-reachable terms rather than on $\mathcal{R}/E$-reachable ones. In the context of verification, $\mathcal{R}/E$-reachable terms that are not $\mathcal{R}$-reachable are not interesting: they are necessarily part of a too coarse approximation. However, in Corollary 112, we show that it is also possible to over-approximate $\mathcal{R}/E$-reachable terms. For some restricted classes of E it is even possible to compute them exactly, as shown in Corollary 111.

Theorem 109 (Completeness) *Let $\mathcal{R}$ be a left-linear TRS, $\mathcal{A}$ be a tree automaton and E be a set of linear equations. If completion terminates on $\mathcal{A}^*_{\mathcal{R},E}$ then*

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

PROOF. First, we prove that $\mathcal{L}(\mathcal{A}^*_{\mathcal{R},E}) \supseteq \mathcal{L}(\mathcal{A})$. By definition of $\mathcal{C}_{\mathcal{R}}(\mathcal{A})$, this algorithm only adds transitions to $\mathcal{A}$ hence, we trivially have $\mathcal{L}(\mathcal{C}_{\mathcal{R}}(\mathcal{A})) \supseteq \mathcal{L}(\mathcal{A})$. Then, thanks to Lemma 88, we know that if $\mathcal{C}_{\mathcal{R}}(\mathcal{A}) \rightsquigarrow^!_E \mathcal{A}'$ then $\mathcal{L}(\mathcal{A}') \supseteq \mathcal{L}(\mathcal{C}_{\mathcal{R}}(\mathcal{A}))$. Hence, by transitivity of $\supseteq$, we know that $\mathcal{L}(\mathcal{A}') \supseteq \mathcal{L}(\mathcal{A})$. This can be successively applied to $\mathcal{A}^1_{\mathcal{R},E}$, $\mathcal{A}^2_{\mathcal{R},E}, \ldots$ so that $\mathcal{L}(\mathcal{A}^*_{\mathcal{R},E}) \supseteq \mathcal{L}(\mathcal{A})$.

Now, the next step of the proof consists in showing that for all term $s \in \mathcal{L}(\mathcal{A})$ if $s \rightarrow_{\mathcal{R}}{}^* t$ then $t \in \mathcal{L}(\mathcal{A}^*_{\mathcal{R},E})$. First, note that by definition of $\rightsquigarrow_E$ final states are preserved, i.e. if q is a final state in $\mathcal{A}$ then if $\mathcal{A} \rightsquigarrow_E \mathcal{A}'$ and q has been renamed in q' then q' is a final state of $\mathcal{A}'$. Hence it is enough to prove that for all term $s \in \mathcal{L}(\mathcal{A}, q)$ if $s \rightarrow_{\mathcal{R}}{}^* t$ then $\exists q' : t \in \mathcal{L}(\mathcal{A}^*_{\mathcal{R},E}, q')$. Because of previous result saying that $\mathcal{L}(\mathcal{A}^*_{\mathcal{R},E}) \supseteq \mathcal{L}(\mathcal{A})$, from $s \in \mathcal{L}(\mathcal{A}, q)$ we obtain that there exists a state q' such that $s \in \mathcal{L}(\mathcal{A}^*_{\mathcal{R},E}, q')$. We know that $s \rightarrow_{\mathcal{R}}{}^* t$ hence, what we have to show is that $t \in \mathcal{L}(\mathcal{A}^*_{\mathcal{R},E}, q')$. By induction on length of $\rightarrow_{\mathcal{R}}{}^*$, we obtain that:

- if length is zero then $s \rightarrow_{\mathcal{R}}{}^* s$ and we trivially have that $s \in \mathcal{L}(\mathcal{A}^*_{\mathcal{R},E}, q')$.
- assume that the property is true for any rewriting derivation of length lesser or equal to n. Now, we prove that it is true for a derivation of length lesser or equal to $n+1$. Assume that we have $s \rightarrow_{\mathcal{R}}{}^n s' \rightarrow_{\mathcal{R}} t$. Using the induction hypothesis, we obtain that $s' \in \mathcal{L}(\mathcal{A}^*_{\mathcal{R},E}, q')$ and it remains to prove that from $s' \rightarrow_{\mathcal{R}} t$ we can deduce that $t \in \mathcal{L}(\mathcal{A}^*_{\mathcal{R},E}, q')$. Since $s' \rightarrow_{\mathcal{R}} t$ we know that there exists a rewrite rule $l \rightarrow r$, a position p and a substitution $\mu : \mathcal{X} \mapsto \mathcal{T}(\mathcal{F})$ such that $s' = s'[l\mu]_p \rightarrow_{\mathcal{R}} s'[r\mu]_p = t$. Since $s' \in \mathcal{L}(\mathcal{A}^*_{\mathcal{R},E}, q')$, $s'[l\mu]_p \rightarrow^*_{\mathcal{A}'} q'$ and be definition of tree automata derivation, we get that there exists a state q'' such that $l\mu \rightarrow^*_{\mathcal{A}^*_{\mathcal{R},E}} q''$ and $l[q'']_p \rightarrow^*_{\mathcal{A}^*_{\mathcal{R},E}} q''$. Let $\mathcal{V}ar(l) = \{x_1, \ldots, x_n\}$, $l = l[x_1, \ldots, x_n]$ and $t_1, \ldots, t_n \in \mathcal{T}(\mathcal{F})$ such that $\mu = \{x_1 \mapsto t_1, \ldots, x_n \mapsto t_n\}$. Since $l\mu = l[t_1, \ldots, t_n] \rightarrow^*_{\mathcal{A}^*_{\mathcal{R},E}} q''$, we know that there exists states $q_1, \ldots, q_n$ such that $\forall i = 1 \ldots n : t_i \rightarrow^*_{\mathcal{A}^*_{\mathcal{R},E}} q_i$ and $l[q_1, \ldots, q_n] \rightarrow^*_{\mathcal{A}^*_{\mathcal{R},E}} q''$. Let $\sigma = \{x \mapsto q_1, \ldots, q_n$. We thus have $l\sigma \rightarrow^*_{\mathcal{A}^*_{\mathcal{R},E}} q''$. Since $\mathcal{A}^*_{\mathcal{R},E}$ is a fixpoint of completion (in particular for $\mathcal{C}_{\mathcal{R}}$), from $l\sigma \rightarrow^*_{\mathcal{A}^*_{\mathcal{R},E}} q''$ we can deduce that $r\sigma \rightarrow^*_{\mathcal{A}^*_{\mathcal{R},E}} q''$. Furthermore, since $\forall i = 1 \ldots n : t_i \rightarrow^*_{\mathcal{A}^*_{\mathcal{R},E}} q_i$, then $r\mu \rightarrow^*_{\mathcal{A}^*_{\mathcal{R},E}} q''$. Finally, since besides of this $s'[q'']_p \rightarrow^*_{\mathcal{A}^*_{\mathcal{R},E}} q'$, we finally have $t = s'[r\mu]_p \rightarrow^*_{\mathcal{A}^*_{\mathcal{R},E}} q'$, hence $t \in \mathcal{L}(\mathcal{A}^*_{\mathcal{R},E}, q')$.

□

Theorem 110 (Correctness) *Let $\mathcal{R}$ be a left-linear TRS, E a set of linear equations and $\mathcal{A}$ a $\mathcal{R}/E$-coherent tree automaton. For any $i \in \mathbb{N}$:*

$$\mathcal{L}(\mathcal{A}^i_{\mathcal{R},E}) \subseteq \mathcal{R}^*_E(\mathcal{L}(\mathcal{A}))$$

PROOF. First, we can prove by induction on i that $\forall i \in \mathbb{N} : \mathcal{A}^i_{\mathcal{R},E}$ is $\mathcal{R}/E$-coherent. The base case is trivial: we know that $\mathcal{A}$ is $\mathcal{R}/E$-coherent. What remains to be proved is that, assuming that $\mathcal{A}^i_{\mathcal{R},E}$ is $\mathcal{R}/E$-coherent, then so is $\mathcal{A}^{i+1}_{\mathcal{R},E}$. By Definition107, we know that if $\mathcal{A}' = \mathcal{C}_{\mathcal{R}}(\mathcal{A}^i_{\mathcal{R},E})$ then $\mathcal{A}' \rightsquigarrow^!_E \mathcal{A}^{i+1}_{\mathcal{R},E}$. Since $\mathcal{A}^i_{\mathcal{R},E}$ is coherent, then so is $\mathcal{A}' = \mathcal{C}_{\mathcal{R}}(\mathcal{A}^i_{\mathcal{R},E})$ by Lemma 106. Finally, using repeated applications of Theorem 97 on $\mathcal{A}' \rightsquigarrow^!_E \mathcal{A}^{i+1}_{\mathcal{R},E}$, $\mathcal{R}/E$-coherence of $\mathcal{A}'$ implies $\mathcal{R}/E$-coherence of $\mathcal{A}^{i+1}_{\mathcal{R},E}$. This concludes the proof that $\forall i \in \mathbb{N} : \mathcal{A}^i_{\mathcal{R},E}$ is $\mathcal{R}/E$-coherent.

What remains to be proved is that terms recognized by states of $\mathcal{A}^i_{\mathcal{R},E}$ are reachable from terms recognized by $\mathcal{A}$ using $\mathcal{R}/E$. Let $\mu = \mu_i \circ \ldots \circ \mu_1$ be the combination of all renamings applied by all completion steps up to $\mathcal{A}^i_{\mathcal{R},E}$. We now prove the following theorem:

$$\forall q \in \mathcal{A} : \mathcal{L}(\mathcal{A}^i_{\mathcal{R},E}, q\mu) \subseteq \mathcal{R}^*_E(\mathcal{L}(\mathcal{A}, q))$$

Since $\mathcal{A}$ is $\mathcal{R}/E$-coherent for any state q in $\mathcal{A}$ we know that there exists a representative s such that $s \rightarrow^{\not{e}*}_{\mathcal{A}} q$. By Definition103, we know that each transition occurring in $\mathcal{A}$ will occur in $\mathcal{C}_{\mathcal{R}}(\mathcal{A})$. Furthermore, by repeated applications of Lemma 86, if $\mathcal{C}_{\mathcal{R}}(\mathcal{A}) \rightsquigarrow^!_E \mathcal{A}_1$ and μ_1 is the renaming such that $\mathcal{A}_1 = \mathcal{C}_{\mathcal{R}}(\mathcal{A})\mu_1$, we get that $s \rightarrow^{\not{e}*}_{\mathcal{A}^1_{\mathcal{R},E}} q\mu_1$. We can proceed similarly up to $\mathcal{A}^i_{\mathcal{R},E}$ and obtain that $s \rightarrow^{\not{e}*}_{\mathcal{A}^i_{\mathcal{R},E}} q\mu$ where $\mu = \mu_i \circ \ldots \circ \mu_1$. Now since $\mathcal{A}^i_{\mathcal{R},E}$ is $\mathcal{R}/E$-coherent, we know that any term $t \rightarrow^*_{\mathcal{A}^i_{\mathcal{R},E}} q\mu$ is such that $s \rightarrow_{\mathcal{R}/E} t$ where $s \in \mathcal{L}(\mathcal{A}, q)$. This ends the proof. □

Computing exactly or over-approximating $\mathcal{R}/E$-reachable terms may be interesting for automated deduction purposes. Using an over-approximation, we may prove for instance that a term is not reachable by rewriting with $\mathcal{R}$ modulo an equational theory E like associativity and commutativity. The first corollary shows that, on some restricted classes of E, if the completion terminates then the resulting automaton recognizes exactly the set of $\mathcal{R}/E$-reachable terms. The second corollary is for general sets of linear equations E and concerns only over-approximations of $\mathcal{R}/E$-reachable terms.

Corollary 111 *Let $\mathcal{R}$ be a left-linear TRS, E be a set of linear equations such that $\forall l = r \in E : \mathcal{V}ar(l) = \mathcal{V}ar(r)$ and $\mathcal{A}$ be a $\mathcal{R}/E$-coherent tree automaton. If completion terminates on $\mathcal{A}^*_{\mathcal{R}\cup\overleftrightarrow{E},E}$ then*

$$\mathcal{L}(\mathcal{A}^*_{\mathcal{R}\cup\overleftrightarrow{E},E}) = \mathcal{R}^*_E(\mathcal{L}(\mathcal{A}))$$

PROOF. Since E is linear and $\forall l = r \in E : \mathcal{V}ar(l) = \mathcal{V}ar(r)$ then $\overleftrightarrow{E}$ is a valid left-linear TRS. Thus, we can instantiate Theorem 109 with $\mathcal{R} \cup \overleftrightarrow{E}$ and obtain that: $\mathcal{L}(\mathcal{A}^*_{\mathcal{R}\cup\overleftrightarrow{E},E}) \supseteq (\mathcal{R} \cup \overleftrightarrow{E})^*(\mathcal{L}(\mathcal{A}))$. We clearly have $(\mathcal{R} \cup \overleftrightarrow{E})^*(\mathcal{L}(\mathcal{A})) = \mathcal{R}^*_E(\mathcal{L}(\mathcal{A})$, we get that $\mathcal{L}(\mathcal{A}^*_{\mathcal{R}\cup\overleftrightarrow{E},E}) \supseteq \mathcal{R}^*_E(\mathcal{L}(\mathcal{A}))$. To obtain the inclusion in the other way, we can similarly instantiate Theorem 110 with $\mathcal{R}\cup\overleftrightarrow{E}$ and get that $\mathcal{L}(\mathcal{A}^*_{\mathcal{R}\cup\overleftrightarrow{E},E}) \subseteq (\mathcal{R}\cup\overleftrightarrow{E})^*_E(\mathcal{L}(\mathcal{A}))$. Note that $(\mathcal{R} \cup \overleftrightarrow{E})^*_E(\mathcal{L}(\mathcal{A})) = (\mathcal{R} \cup \overleftrightarrow{E})^*(\mathcal{L}(\mathcal{A}))$ which is equal to $\mathcal{R}^*_E(\mathcal{L}(\mathcal{A}))$. Hence, $\mathcal{L}(\mathcal{A}^*_{\mathcal{R}\cup\overleftrightarrow{E},E}) \subseteq \mathcal{R}^*_E(\mathcal{L}(\mathcal{A}))$ and we get the result. □

Note that the restriction on E saying that $\forall l = r \in E : \mathcal{V}ar(l) = \mathcal{V}ar(r)$ is necessary. Otherwise, the oriented system $\overleftrightarrow{E}$ is no longer a valid TRS and this raises problems w.r.t. R/E-coherence. We can illustrate this on a simple equation, say $E = \{x * 0 = 0\}$. We have $\overleftrightarrow{E} = \{x * 0 \to 0, 0 \to x * 0\}$. The rule $0 \to x * 0$ is not a valid TRS rule since $\mathcal{V}ar(0) \not\supseteq \mathcal{V}ar(x * 0)$. However, completion can be performed on such rules using a technique close to (Jacquemard, 1996). The idea is to add a special state, say $q_{\mathcal{T}(\mathcal{F})}$, and a set of transitions to the initial automaton $\mathcal{A}$, such that $\forall t \in \mathcal{T}(\mathcal{F}) : t \to_{\mathcal{A}}{}^* q_{\mathcal{T}(\mathcal{F})}$. Then it is enough to replace all variables occurring only in the right-hand side of a rule by $q_{\mathcal{T}(\mathcal{F})}$ and perform a usual completion on $\mathcal{A}$ extended with $q_{\mathcal{T}(\mathcal{F})}$ and the related transitions. During completion, if there is a critical pair between the rewrite rule $0 \to q_{\mathcal{T}(\mathcal{F})} * 0$ and a transition $0 \to_{\mathcal{A}} q$, it can be solved by adding: $q_{\mathcal{T}(\mathcal{F})} * 0 \to q'$ and $q' \to q$. However, adding transitions to $\mathcal{A}$ such that $\forall t \in \mathcal{T}(\mathcal{F}) : t \to_{\mathcal{A}}{}^* q_{\mathcal{T}(\mathcal{F})}$ prevents the tree automaton $\mathcal{A}$ from being $\mathcal{R}/E$-coherent in general. Indeed, for all $t_1, t_2 \in \mathcal{T}(\mathcal{F})$, t_1, t_2 are recognized by $\mathcal{A}$ without epsilon transitions, i.e. $t_1 \to_{\mathcal{A}}^{\not\varepsilon *} q_{\mathcal{T}(\mathcal{F})}$ and $t_2 \to_{\mathcal{A}}^{\not\varepsilon *} q_{\mathcal{T}(\mathcal{F})}$. For this extended $\mathcal{A}$ to be $\mathcal{R}/E$-coherent, it would be necessary that $\forall t_1, t_2 \in \mathcal{T}(\mathcal{F}) : t_1 =_E t_2$, which not true in general. Without $\mathcal{R}/E$-coherence it is not possible to prove correctness but completeness is still valid for general linear system of equations as it is shown in the following corollary.

Let us call *extended* completion the completion described above, dealing with rules $l \to r$ such that $\mathcal{V}ar(r) \not\subseteq \mathcal{V}ar(l)$ using a $q_{\mathcal{T}(\mathcal{F})}$ state recognizing $\mathcal{T}(\mathcal{F})$.

Corollary 112 *Let $\mathcal{R}$ be a left-linear TRS, $\mathcal{A}$ be a tree automaton and E be a set of linear equations. Let $\overleftrightarrow{E} = \{l \to r, r \to l \mid l = r \in E\}$ and $\mathcal{B} = \mathcal{A} \cup \{f(q_{\mathcal{T}(\mathcal{F})}, \ldots, q_{\mathcal{T}(\mathcal{F})}) \to q_{\mathcal{T}(\mathcal{F})} \mid f \in \mathcal{F}\}$. If extended completion terminates on $\mathcal{B}^*_{\mathcal{R} \cup \overleftrightarrow{E}, E}$ then*

$$\mathcal{L}(\mathcal{B}^*_{\mathcal{R} \cup \overleftrightarrow{E}, E}) \supseteq \mathcal{R}^*_E(\mathcal{L}(\mathcal{A}))$$

PROOF. It is easy to see that Theorem 109 can be lifted to the case of extended completion. Given a rewrite rule of the form $l \to C[x]$ where x does not occur in l, if a term s is rewritten using this rule then $s|_p = l\mu$ and $s \to_{\mathcal{R}} s[C[t]\mu]|_p$, where t is possibly any term of $\mathcal{T}(\mathcal{F})$ and $\mu : \mathcal{X} \mapsto \mathcal{T}(\mathcal{F})$. During completion, as shown in Theorem 109, from $s \to_{\mathcal{R}} s[C[t]\mu]|_p$ we can deduce that there necessarily exists a state q and a substitution $\sigma : \mathcal{X} \mapsto \mathcal{Q}$ such that $l\mu \to_{\mathcal{B}}^* l\sigma \to_{\mathcal{B}}^* q$. Extended completion will produce a new automaton $\mathcal{B}'$ containing transitions such that $C[q_{\mathcal{T}(\mathcal{F})}]\sigma \to_{\mathcal{B}'}^* q$. Since $\mathcal{B}'$ also contains transitions recognizing any term of $\mathcal{T}(\mathcal{F})$ in $q_{\mathcal{T}(\mathcal{F})}$, we have in particular that $C[t]\mu \to_{\mathcal{B}'}^* \mathcal{C}[q_{\mathcal{T}(\mathcal{F})}]\sigma \to_{\mathcal{B}'}^* q$. The proof that $s[C[t]\mu]|_p$ is recognized by $\mathcal{B}'$ can be finished in the same way as for Theorem 109. Thus, this theorem lifted to extended completion can be instantiated by $\mathcal{R} \cup \overleftrightarrow{E}$ and $\mathcal{B}$ and we obtain that: $\mathcal{L}(\mathcal{B}^*_{\mathcal{R} \cup \overleftrightarrow{E}}) \supseteq (\mathcal{R} \cup \overleftrightarrow{E})^*(\mathcal{L}(\mathcal{B}))$. We clearly have $(\mathcal{R} \cup \overleftrightarrow{E})^*(\mathcal{L}(\mathcal{B})) = \mathcal{R}^*_E(\mathcal{L}(\mathcal{B})$, thus we get that $\mathcal{L}(\mathcal{B}^*_{\mathcal{R} \cup \overleftrightarrow{E}, E}) \supseteq \mathcal{R}^*_E(\mathcal{L}(\mathcal{B}))$. Finally since $\mathcal{B}$ is obtained from $\mathcal{A}$ by a simple extension of its set of transitions, we trivially have $\mathcal{L}(\mathcal{B}) \supseteq \mathcal{L}(\mathcal{A})$. Hence $\mathcal{R}^*_E(\mathcal{L}(\mathcal{B})) \supseteq \mathcal{R}^*_E(\mathcal{L}(\mathcal{A}))$ and finally $\mathcal{L}(\mathcal{B}^*_{\mathcal{R} \cup \overleftrightarrow{E}, E}) \supseteq \mathcal{R}^*_E(\mathcal{L}(\mathcal{A}))$. $\square$

3.3 Comparison with other work

In this section we show how standard completion and equational completion compares with other work. In the first part, we show that most of the known classes of TRS, presented in Section 2.1.1, for which the set of reachable terms is regular are covered by the standard completion algorithm. In other words, for all those classes, the standard completion algorithm terminates on a tree automaton recognizing *exactly* the set of reachable terms. In the second part, we show that this result is still valid for the equational completion algorithm. Then, in the third part, we compare the tree automata completion framework with the equational abstraction due to Meseguer et al.. Finally, in the fourth part, we show how classical static analysis techniques based on regular languages can be implemented using the equational completion algorithm.

3.3.1 Recognizing regular classes with standard completion

If we consider the standard completion, we have proven in Theorem 76 that if the abstraction function α is injective, coherent with $\mathcal{R}$ and $\mathcal{A}$ that also fulfill the left and right coherence conditions then if completion terminates on $\mathcal{A}^*_{\mathcal{R},\alpha}$ then $\mathcal{L}(\mathcal{A}^*_{\mathcal{R},\alpha}) = \mathcal{R}^*(\mathcal{L}(\mathcal{A}))$. Moreover, in Lemma 82, we have shown that if $\mathcal{R}an(\alpha)$ is finite then completion terminates. Note that for injective abstraction functions, the range is finite if the domain is. Hence, it is enough to prove that completion produce a finite number of distinct transitions to be normalized. Now we give alternative algorithm and proofs of regularity of $\mathcal{R}^*(\mathcal{S})$ for some of the classes described in Section 2.1.1. In some cases, even if the TRS is not left-linear, the left-coherence condition can be guaranteed by determinizing the tree automaton after each completion step, as for **RL-M** class. However, for more complex classes as **RL-FPO**, it is possible to do the same but, then, it is no longer possible to guarantee that $\mathcal{R}an(\alpha)$ remains finite. This is the reason why the **RL-FPO** is not covered by completion. Some intuitions about (Takai et al., 2000) for dealing with **RL-FPO** and alternatives to determinisation for dealing with possibly non left-linear TRSs are given in Section 4.4.1. Let $\mathcal{S}$ be a regular language and $\mathcal{R}$ a TRS belonging to one of the following classes:

G for ground TRS (Dauchet and Tison, 1990; Brainerd, 1969), we use corollary 81 (ground TRSs are linear) and an injective abstraction α with a finite domain $\{r|_p \mid l \to r \in \mathcal{R} \text{ and } p \in \mathcal{P}os(r) \setminus \{\epsilon\}\}$. We can restrict α to this finite domain since in every new transition $f(t_1, \ldots, t_n) \to q$ added by the completion, $f(t_1, \ldots, t_n)$ is necessarily ground and is a right-hand side of a rule of $\mathcal{R}$. So it is enough to normalize $t_1, \ldots, t_n$ and all their subterms to normalize the transition. Hence, in α for every rule $l \to r$, every strict subterm of r is mapped to a new state. Since the domain is finite, so is the range and completion terminates.

RL-M for right-linear ans monadic TRS (Salomaa, 1988), we use theorem 76 and an abstraction function α with an empty domain which satisfy the coherence property w.r.t. $\mathcal{R}$ and $\mathcal{A}$ and is also trivially injective. The domain of α is empty since every new transition produced by the completion is of the form $f(q_1, \ldots, q_n) \to q$ where $q_1, \ldots, q_n$ are states or of the form $q \to q'$. None of these forms needs to be normalized. Assume that after each completion step, we determinize the completed

automaton $\mathcal{A}^n_{\mathcal{R},\alpha}$. Since the domain of α is empty, completion ends on $\mathcal{A}^*_{\mathcal{R},\alpha}$ which is determinized. Thanks to determinization of the last completion step left-coherence condition is trivially satisfied and since the TRS is right linear, this is also the case for right-coherence condition.

L-SM for linear and semi-monadic TRS (Coquidé et al., 1991), as in the ground case we define α as an injective function on the finite domain: $\{r|_p \mid l \to r \in \mathcal{R}$ and $p \in \mathcal{P}os_{\mathcal{F}}(r)\}$. Similarly, we can restrict to this finite domain since in every new transition $f(t_1, \ldots, t_n) \to q$ added by the completion, t_i is either a ground term (and can be normalized by a single state) or is itself a state and thus does not need normalization.

L-G^{-1} for linear and inversely growing TRS (Jacquemard, 1996), recall that it means that every right-hand side is either a variable, or a term $f(t_1, \ldots, t_n)$ where $f \in \mathcal{F}$, $\mathcal{A}r(f) = n$, and $\forall i = 1, \ldots, n$, t_i is a variable, a ground term, or a term whose variables do not occur in the left-hand side. For this class, the proof and abstraction function is similar to the linear and semi-monadic case except for variables occurring in the right-hand side but not in the left-hand side. For those variables, it is enough to substitute them by a specific state $q_{\mathcal{T}(\mathcal{F})}$ (which recognize $\mathcal{T}(\mathcal{F})$) and add the set of transitions $\{f(q_{\mathcal{T}(\mathcal{F})}, \ldots, q_{\mathcal{T}(\mathcal{F})}) \to q_{\mathcal{T}(\mathcal{F})} \mid f \in \mathcal{F},\ \mathcal{A}r(f) = n\}$ to the transitions of $\mathcal{A}$. As in the linear and semi-monadic case, we define α as an injective function on the finite domain: $\{r|_p \mid l \to r$ and $p \in \mathcal{P}os_{\mathcal{F}}(r)\}$ where variables in r not occurring in l are substituted by $q_{\mathcal{T}(\mathcal{F})}$.

L-GFPO for linear generalized finite path overlapping TRS (Takai, 2004), we use theorem 76 and we construct injective abstraction function α such that $\mathcal{R}an(\alpha) \cap \mathcal{Q} = \emptyset$. Furthermore since $\mathcal{R}$ is linear, left-coherence and right-coherence are satisfied. As shown in Section 2.1.2, (Takai, 2004; Takai et al., 2000) normalize and add transitions to a tree automaton using the addtrans procedure. For all $f \in \mathcal{F}$ and packed states $q_1, \ldots, q_n$, this procedure normalize any configuration $f(q_1, \ldots, q_n)$ by a packed state $\langle f(q_1, \ldots, q_n)\rangle$. In other words, it creates an injection, say β, associating each configuration to a single packed state. Besides to this, the termination theorem of (Takai, 2004) shows that using this normalization technique the tree automata construction only produces a finite number of new packed states. In other words, the injection β maps all the configurations to a finite number of packed states, i.e. its range is finite. Let us call P the range of β. To prove termination of tree automata completion on this class, it is enough to define α as β. More formally, let $Q' = \{\langle q\rangle \mid q \in \mathcal{Q}\}$. We can define α by $\forall q_1, \ldots, q_n \in Q' \cup P : \alpha(f(q_1, \ldots, q_n) = \langle f(q_1, \ldots, q_n)\rangle$. Since $\alpha = \beta$ and $\mathcal{R}an(\beta) = P$ is finite, so is the range of α, i.e. $\mathcal{R}an(\alpha) = P$. Note also that $\mathcal{R}an(\alpha) \cap Q = P \cap Q = \emptyset$.

Constructor based (Réty, 1999) For this particular case, there is also a restriction on the initial language $\mathcal{S} = \{t\sigma\}$ where $t \in \mathcal{T}(\mathcal{F}, \mathcal{X})$ is linear and $\sigma : \mathcal{X} \mapsto \mathcal{T}(\mathcal{C})$. Let $\mathcal{A} = \langle \mathcal{F}, \mathcal{Q}, \mathcal{Q}_f, \Delta\rangle$ be the tree automaton recognizing $\mathcal{S}$. In this particular case, our aim is more to give an alternative algorithm rather than a proof of regularity. As in (Réty, 1999), we focus on the algorithm for left and right-linear TRSs since the left-coherence restriction can be discarded using determinization of tree automata (as

in the right-linear and monadic case). We use theorem 76 and an injective abstraction function α such that $\mathcal{R}an(\alpha) \cap \mathcal{Q} = \emptyset$. Furthermore since $\mathcal{R}$ is linear, left-coherence and right-coherence are satisfied. Now, let us prove that the domain of α is finite. Let $Q_{t\sigma}$ be the finite set of states necessary to normalize deterministically $t\sigma$, Q_{arg} be the set of states necessary to normalize the ground subterms of the right-hand sides of the rules and Δ_{arg} the related set of transitions. In (Réty, 1999), it is shown that for every defined symbol of t, for every substitution $\delta : \mathcal{X} \mapsto \mathcal{T}(\mathcal{F})$, for every rewriting $l\delta \rightarrow_{\mathcal{R}} r\delta$, there exists a substitution $\sigma : \mathcal{X} \mapsto Q_{t\sigma} \cup Q_{arg}$ such that $l\delta \rightarrow_{\Delta \cup \Delta_{arg}} l\sigma$ and $r\delta \rightarrow_{\Delta \cup \Delta_{arg}} r\sigma$. Hence, every critical pair encountered during the completion is of the form: $l\sigma \rightarrow_{\mathcal{A}} q$ and $l\sigma \rightarrow_{\mathcal{R}} r\sigma$ with $\sigma : \mathcal{X} \mapsto Q_{t\sigma} \cup Q_{arg}$. Since the number of defined symbols of t is finite, since $Q_{t\sigma} \cup Q_{arg}$ is finite, then so is the set of every possible critical pair and so is the domain of α.

In this last case, we did not give an explicit definition of α. The good news, and this is one of the main interest of tree automata completion algorithm in practice, is that it is useless to define α a priori since it can be constructed automatically *during* completion. For all the above classes, since the domain of α is finite and since α is injective, we can construct an injective α function "on-the-fly" by associating a new state to every subterm occurring during the normalization of a new transition. This leads to a fully automatic and terminating completion algorithm covering all the decidable classes we summed up here. We call this normalization strategy the *exact* normalization strategy.

Definition 113 (Exact normalization strategy) *The* exact normalization strategy *consists in building α incrementally during completion. We start from $\alpha = \emptyset$. Then, during completion, let $\mathcal{A} = \langle \mathcal{F}, \mathcal{Q}, \mathcal{Q}_f, \Delta \rangle$ be the automaton to complete and c a new configuration to normalize.*

- *If $c \in \mathcal{D}om(\alpha)$ then we normalize c with $\alpha(c)$, otherwise*
- *we create a new state q such that $q \notin \mathcal{R}an(\alpha)$ and $q \notin \Delta$, add the association $c \mapsto q$ in α and normalize c with q*

$\diamond$

Note that the approximation function α produced by the exact normalization strategy is coherent and injective by construction. We can thus benefit of the Theorem 76 and be sure that, when using the exact normalization strategy, completion produces a tree automaton recognizing *exactly* reachable terms. Furthermore, we have seen that completion is guaranteed to terminate for all the above decidable classes. In those cases, the completed automaton $\mathcal{A}^*_{\mathcal{R},\alpha}$ recognizes $\mathcal{R}^*(\mathcal{S})$ if $\mathcal{R}$ is linear or if $\mathcal{R}$ is right-linear and $\mathcal{R}$ and $\mathcal{A}^*_{\mathcal{R},\alpha}$ satisfy the left-coherence condition. Note that all the TRSs of the above classes are at least right-linear and thus right-coherence condition is fulfilled. The left-coherence condition is implied either by left-linearity or by successive determinization of each completion steps.

Now, for the two classes **IOS-LTS** and **WOS** we do not give an alternative proof of regularity but show how to compute the set of reachable terms using standard completion with normalization rules.

L-IOSLT (Seki et al., 2002) For this class, as for the **L-GFPO** class, we can simulate the application of the addtrans procedure by defining an injection α by $\alpha(f(q_1, \ldots, q_n)) = \langle f(q_1, \ldots, q_n) \rangle$. What remains to be done is to simulate the construction of product states. Recall that for all transition $p(g(t_1, \ldots, t_n)) \to q$ to be added, a *unique* product state $[q, p]$ is created so as to recognize the term $g(t_1, \ldots, t_n)$. As a result the above transition is normalized into the set $\{p([q, p]) \to q, g(t_1, \ldots, t_n) \to [q, p]\}$ and terms $t_1, \ldots, t_n$ are normalized, if necessary, using the addtrans procedure. The construction of those product states can be implemented using the following set of normalization rules:

$$\{[p(x) \to z] \to [x \to prod(z, p)] \mid p \in P\}$$

where $prod : \mathcal{Q} \times P \mapsto \mathcal{Q}$ is a specific new state symbol of arity 2.

WOS (Bouajjani and Touili, 2002) Recall that **L-IOSLT** are equivalent to linear bottom-up tree transducers. Furthermore, it is shown in (Bouajjani and Touili, 2002) that the set of reachable terms by a **WOS** TRS can be obtained using by computing closures $R^*_{T_i}$ of a finite number of tree transducers R_{T_i} for $i = 1 \ldots n$. Since, each of these tree transducers can be encoded into a **L-IOSLT** whose closure can be computed as above, **WOS** reachable terms can be obtained by repeated applications of completion.

Finally, beside to the classes described above, completion with exact normalization strategy can compute exactly sets of reachable terms for cases that are not included in the known decidable classes. A very simple example is the TRS $\mathcal{R} = \{f(g(x)) \to g(f(x))\}$ and the initial language $\mathcal{S} = \{h^*(f(g^*(a)))\}$. First, this TRS does not preserve regularity of all set $\mathcal{S}$. For instance, if we choose the initial set $\mathcal{S}' = \{(fg)^*(a)\}$ the set $\mathcal{R}^*(\mathcal{S}')$ is not regular. This is due to the fact that $\mathcal{R}^!(\mathcal{S}') = \mathcal{R}^*(\mathcal{S}') \cap IRR(\mathcal{R})$ and $IRR(\mathcal{R})$ is regular. If $\mathcal{R}^*(\mathcal{S}')$ was regular then so would be $\mathcal{R}^!(\mathcal{S}')$. However, $\mathcal{R}^!(\mathcal{S}') = \{g^n(f^n(a)) \mid n \in \mathbb{N}\}$ which is not regular and, thus, $\mathcal{R}^*(\mathcal{S}')$ is not regular. Hence, this TRS is outside of all the classes that do not impose restrictions on the initial set of terms, i.e. all of them except the class **Constructor based**. This particular example is also outside of the **Constructor based** class because the initial set is not of the form $\{t\sigma\}$ where t is a linear term. However, this TRS is linear and completion terminates with an injective abstraction function automatically built with the exact completion strategy. Thus, we have a proof of the regularity of $\mathcal{R}^*(\mathcal{S})$ and it is recognized by the completed automaton $\mathcal{A}^*_{\mathcal{R},\alpha}$.

Another simple example is the TRS $\mathcal{R} = \{f(x) \to f(g(f(x)))\}$ and the initial language $\mathcal{S} = \{f(a)\}$. This TRS is outside of all the regular classes we survey in Chapter 2. In particular, it is outside the **L-GFPO** class because its GFPO-graph has only one node and a looping edge of weight 1 on it. This is due to the fact that $f(g(f(x)))$ properly sticks out of $f(x)$. It is also outside of the **Constructor based** class because the right hand side of the rewrite rule has nested function symbol: f. However, again, completion terminates with the exact completion strategy and ends on a tree automaton recognizing $\mathcal{R}^*(\mathcal{S}) = \{(fg)^* f(a)\}$.

3.3.2 Recognizing regular classes with equational completion

In previous Section 3.3.1, we showed that many of the known regular classes were covered by the standard tree automata completion algorithm. In this section, we lift the recognizability results to the case of the equational completion algorithm. We are going to use a similar exact normalization strategy and an empty set of equations. What we need to show is that the epsilon-based completion step does not jeopardize termination of completion on regular classes of Section 3.3.1. Here, we only consider the classes for which no normalization rules were necessary, i.e. all the classes except the **IOS-LTS** and **WOS** classes.

First, we can remark that the exact normalization strategy (Definition 113 of Section 3.3.1) is compatible with Definition 99 of normalization for equational completion. We can implement the exact normalization strategy in $Norm_\Delta$ as follows. Let $\Delta_0, \Delta_1, \ldots$ be the respective sets of transitions of automata $\mathcal{A}^0, \mathcal{A}^1_\mathcal{R}, \ldots$ obtained by equational completion. Hence, Δ_0 is the set of transitions of the initial automaton $\mathcal{A}$ before completion. At step k of completion, in the Definition 99 of normalization, instead of choosing arbitrarily q, we define q as the right-hand side of a transition of the form $f(q'_1, \ldots, q'_n) \to q$ if it exists in $\Delta_k \setminus \Delta_0$ and as a new state otherwise. This implements the exact normalization where we use the set $\Delta_k \setminus \Delta_0$ in place of the abstraction function α. Let α_k be the definition of α at step k of completion. We define α_k in the following way: $\forall c \in \mathcal{T}(\mathcal{F} \cup \mathcal{Q}) \setminus \mathcal{Q} : \forall q \in \mathcal{Q} : \alpha_k(c) = q$ if $c \to q \in \Delta_k \setminus \Delta_0$. Note that, α is a function because $\Delta_k \setminus \Delta_0$ is deterministic by construction. This is due to the fact that each transition of this set is added by $Norm_\Delta$ and that, when it implements the exact completion strategy, this function normalize each configuration by a *distinct* state.

Thus, using the Theorem 110 and the Theorem 109, if equational completion of $\mathcal{A}$ for a TRS $\mathcal{R}$ and $E = \emptyset$ terminates on a tree automaton $\mathcal{A}^*_{\mathcal{R},E}$ then we have $\mathcal{L}(\mathcal{A}^*_{\mathcal{R},E}) = \mathcal{R}^*(\mathcal{L}(\mathcal{A}))$. Since $E = \emptyset$, in the following we denote each equational completion step $\mathcal{A}^i_{\mathcal{R},E}$ by $\mathcal{A}^i_\mathcal{R}$. What remains to be proven is that equational completion terminates for the same TRS classes than standard completion.

Theorem 114 *With the exact normalization strategy, if the standard completion terminates then so is the equational completion with $E = \emptyset$.*

PROOF. Since $E = \emptyset$, no simplification step is performed between each completion steps. As a results, the only difference between standard and equational completion is how critical pairs are solved, i.e. use

$$\begin{tikzcd} l\sigma \arrow[r, "\mathcal{R}"'] \arrow[d, "\mathcal{A}^i_\mathcal{R}"'] & r\sigma \arrow[d, "\mathcal{A}^{i+1}_\mathcal{R}"] \\ q & q' \arrow[l, "\mathcal{A}^{i+1}_\mathcal{R}"] \end{tikzcd} \quad \text{instead of} \quad \begin{tikzcd} l\sigma \arrow[r, "\mathcal{R}"'] \arrow[d, "\mathcal{A}^i_\mathcal{R}"', "*"] & r\sigma \arrow[dl, bend left, "*"] \\ q & \end{tikzcd} \quad \mathcal{A}^{i+1}_\mathcal{R}$$

Each equational completion step potentially introduce a new state q'. Now, we prove that adding those new states only add a finite number of critical pairs to consider and, thus, that equational completion terminates. Let A be a tree automaton and $\mathcal{R}$ a TRS such that standard completion terminates on $\mathcal{A}$, $\mathcal{R}$. Let $\mathcal{B}^0_\mathcal{R} = \mathcal{A}, \mathcal{B}^1_\mathcal{R}, \ldots$ be the standard completion steps and $\mathcal{A}^0_\mathcal{R} = \mathcal{A}, \mathcal{A}^1_\mathcal{R}, \ldots$ be the equational completion steps. Let P^A_k (resp. P^B_k) be the union of all the substitutions $\sigma : \mathcal{X} \mapsto \mathcal{Q}$ of all critical pairs $(l \to r, \sigma, q)$

encountered during the equational (resp. standard) completion of $\mathcal{A}$ up the $\mathcal{A}^k_{\mathcal{R}}$ (resp $\mathcal{B}^k_{\mathcal{R}}$). Since standard completion terminates, we know that the fixpoint $\mathcal{B}^*_{\mathcal{R}}$ exists and that P^B_* is finite. Now, we aim at proving that $P^A_* = P^B_*$ and thus is finite. We prove that those two sets are equal for any number $k \in \mathbb{N}$ of completion steps. This proof is done by induction on k.

- For the first completion step, i.e. producing $\mathcal{A}^1_{\mathcal{R}}$ from $\mathcal{A}^0$, we trivially have $P^A_1 = P^B_1$. Indeed, since both algorithm compute the substitutions from the same automaton $\mathcal{A}^0$, they have the same value.

- Now, we assume that the property $P^A_k = P^B_k$ is true up the k-th completion step. Note that, up to completion step k, we can assume that all new transitions, found by both completion algorithm, are normalized in the same way, with the same state numbering. For every $l \to r \in \mathcal{R}$, if there is a similar critical pair $l\sigma \to^*_{\mathcal{B}^i_{\mathcal{R}}} q$ and $l\sigma \to^*_{\mathcal{A}^i_{\mathcal{R}}} q'$ with $i \leq k$, then we assume that all the subterms of $r\sigma$ will be normalized by the same states in $\mathcal{B}^{i+1}_{\mathcal{R}}$ and $\mathcal{A}^{i+1}_{\mathcal{R}}$.

  Now, we consider the $k+1$ completion step, i.e. we compute $\mathcal{A}^{k+1}_{\mathcal{R}}$ from $\mathcal{A}^k_{\mathcal{R}}$ and $\mathcal{B}^{k+1}_{\mathcal{R}}$ from $\mathcal{B}^k_{\mathcal{R}}$. Assume that after completion step $k+1$ there exists a state p' not occurring in any substitution of P^B_{k+1} but occurring in the range of a substitution $\sigma' \in P^A_{k+1} \setminus P^B_{k+1}$. Since up to k-th completion step all substitutions and normalizations of both completion algorithm coincide, the state p' necessarily appeared in a completion step i $(1 \leq i < k)$ to recognize the right-hand side of a rewrite rule, i.e.

$$\begin{tikzcd} l\sigma \arrow[r, "\mathcal{R}"'] \arrow[d, "\mathcal{A}^i_{\mathcal{R}}"'] & r\sigma \arrow[d, "\mathcal{A}^{i+1}_{\mathcal{R}}"] \\ q & p' \arrow[l, "\mathcal{A}^{i+1}_{\mathcal{R}}"] \end{tikzcd}$$

  Note that in the above transitions p' appears only in the left-hand side of an epsilon transition $p' \to q$ and, thus, cannot appear in the range of the substitution of a non trivial critical pair. Indeed, a state r appears in a substitution of a non trivial critical pair only if it appears in the left-hand side of regular transitions $f(\ldots r \ldots) \to r'$ of Δ. Hence, for p' to appear in a regular transition $f(\ldots p' \ldots) \to r'$, it needs to have been used to normalize a subterm t in $f(\ldots t \ldots)$. Besides to this, for p' to appear in a substitution, this imposes that there exists a rewrite rule whose left-hand side contains a subterm of the form $f(\ldots x \ldots)$. Moreover, we also know that this subterm matches the transition $f(\ldots p' \ldots) \to r'$. Since the normalization strategy is deterministic and since, from the above critical pair solving, we know that $r\sigma \to^*_{\mathcal{A}^{i+1}_{\mathcal{R}}} p'$, we necessarily have $t = r\sigma$. For $f(\ldots r\sigma \ldots)$ to have been normalized, this means that during completion step j $(1 \leq j < k)$ another critical pair of the form

$$\begin{array}{ccc} s & \xrightarrow[\mathcal{R}]{} & C[f(\ldots r\sigma\ldots)] \\ {\scriptstyle \mathcal{A}^j_{\mathcal{R}}}\downarrow & & \downarrow{\scriptstyle \mathcal{A}^{j+1}_{\mathcal{R}}} \\ q_1 & \xleftarrow[\mathcal{A}^{j+1}_{\mathcal{R}}]{} & p_1 \end{array}$$

has been solved. Moreover, $f(\ldots r\sigma\ldots)$ has been normalized into $r\sigma \to p'$ and $f(\ldots p'\ldots) \to r'$. Since the two completion algorithm produce tree automata recognizing the same languages, $C[f(\ldots r\sigma\ldots)]$ is also recognized by $\mathcal{B}^j_{\mathcal{R}}$. Since, up to completion step k, sets of substitutions coincide, we know that a similar critical pair has been solved on tree automaton $\mathcal{B}^j_{\mathcal{R}}$. Hence, a transition of the form $C[f(\ldots r\sigma\ldots)] \to q_2$ has been normalized in order to be added to $\mathcal{B}^j_{\mathcal{R}}$. As mentioned above, we can assume that new transitions produced by both completion algorithm can be normalized using the same state numbering. Thus, the state recognizing $r\sigma$ is similar in $\mathcal{A}^{j+1}_{\mathcal{R}}$ and in $\mathcal{B}^{j+1}_{\mathcal{R}}$, it is thus p'. This proves that the configuration $f(\ldots p'\ldots)$ appears also in $\mathcal{B}^j_{\mathcal{R}}$ and that it can thus be matched by the subterm $f(\ldots x\ldots)$ of the left-hand side of a rule. We can thus build a substitution mapping x to p' in $\mathcal{B}^k_{\mathcal{R}}$. Finally, this proves that p' also appears in a substitution σ' of P^B_{k+1} which is a contradiction. Thus, $P^A_{k+1} = P^B_{k+1}$

Since the set P^B_k of substitutions σ for standard completion finitely reaches a fixpoint, so is P^A_{k+1}. Hence, in equational completion there are only a finite number of possible $r\sigma$ to add. Since we use a deterministic exact normalization strategy, each $r\sigma$ is normalized by a distinct state q'. We thus have only a finite number of critical pairs to solve and this proves that equational completion terminates. □

3.3.3 Equational abstraction

In this section, we compare the equational tree automata completion framework with the equational abstraction technique defined by (Meseguer et al., 2003) and also related to (Takai, 2004). As far as we know the idea of defining abstractions using equations was first introduced in (Genet and Viet Triem Tong, 2002). However, it was more convincingly defined in (Meseguer et al., 2003), followed by (Takai, 2004) and then by (Genet and Rusu, 2009). Though the abstraction principle is the same, it has been put into practice differently in all those works.

(Takai, 2004) defined a completion-like algorithm that is able to deal exactly with the **L-GFPO** class and, if the TRS is not included in this class, performs a widening. In some cases, the widening forces the algorithm to terminate. The widening consists in inferring automatically abstraction transitions (see Section 2.2.2) forcing the creation of loops in the completed tree automaton. (Takai, 2004) says that those abstraction transitions can be seen as the result of the application of abstraction equations. However, this is essentially an intuition and no formal proof is given. In our setting, thanks to the correctness Theorem 110 and under some hypothesis on the tree automaton, we know that the automaton resulting of completion *exactly* recognizes the result of the application of abstraction equations. Nevertheless, unlike (Takai, 2004) our completion framework does not propose any technique

to detect a widening position and automatically build an approximation. It would be interesting to refine the widening technique proposed by Takai to produce automatically a set of approximation equations and to apply them using our equational completion algorithm. Since equational completion can handle any set of linear equations, it can deal with any equation found by Takai algorithm which are necessary of the form $C[C[x]] = C[x]$. Furthermore, in our setting, if completion does not terminate using this automatically generated set of equations, it is possible to enrich it by hand so as to force termination.

As explained in the introduction, we believe that asking a human for approximation equations is reasonable when fully automatic verification techniques fail. Similarly, (Meseguer et al., 2003) assume that we *have* an a priori set of abstraction equations. The objective of the paper is to check by rewriting some properties on the model abstracted by those equations. The properties detailed in this paper are safety and liveness properties. On the one side, safety properties can be seen as unreachability proofs, i.e. "nothing bad happens", and very similar to the properties we are interested in this document. On the other side, liveness properties, i.e. "something good eventually happens" are out of reach of the techniques proposed here. Furthermore, the class of TRS they can use to model programs is more general. They are not restricted to left-linear TRS or left-linear conditional TRS. On the contrary, they theoretically can deal with general conditional TRS modulo associativity, commutativity etc., i.e. the general rewriting framework of the Maude tool (Clavel et al., 2001). Additionally, the abstraction equations are not limited to linear ones but can be non linear and even conditional. The only limitation this framework has, w.r.t. equational tree automata completion, is that it can only handle finite sets of initial terms $\mathcal{S}$.

However, the generality of their framework has a strong counterpart. As explained in Section 2.2.1, given a TRS $\mathcal{R}$, a set of equations E and two terms s and t for proving that $s \not\rightarrow_{\mathcal{R}}{}^{*} t$ they rely on the property $\rightarrow_{\mathcal{R}} \subseteq \rightarrow_{\mathcal{R}/E}$ and prove that $s \not\rightarrow^{*}_{\mathcal{R}/E} t$. Since the Maude framework performs only rewriting, in order to model rewriting with $\mathcal{R}/E$, the set of equations E has to be oriented into a TRS $\mathcal{R}'$ enjoying many properties. Among other properties, the TRS $\mathcal{R}'$ has to be terminating, ground confluent and coherent with $\mathcal{R}$ (see Section 2.2.1). This is a strong restriction on the sets of equations which can be used to perform the abstraction. The equation set you need may not have this property and even if it has the property the proof may be non trivial as it is shown Section 9.3 of (Clavel et al., 2008) or in (Meseguer et al., 2003).

Finally, if the set E has the good properties it remains to prove that $s \not\rightarrow^{*}_{\mathcal{R}/E} t$. For doing this proof in Maude, it is necessary for s to be rewritten by $\mathcal{R}/E$ into a finite number of E-equivalent terms. In other words, $\mathcal{R}/E$ needs to be terminating on s or starting from s there is only a finite number of E-equivalent reachable terms. We now show that, in this case and on the subclasses of TRS we consider, equational completion offers the same termination guarantee.

Lemma 115 *If $\mathcal{R}$ is left-linear and E is linear and $\rightarrow_{\mathcal{R}/E}$ terminates on s then equational completion with $\mathcal{R}$ and E terminates on the tree automaton recognizing s.*

PROOF. Let $\mathcal{A}$ be an epsilon-free deterministic tree automaton obtained by normalizing s. The tree automaton $\mathcal{A}$ is $\mathcal{R}/E$-coherent by construction. In a deterministic $\mathcal{R}/E$-coherent tree automaton, each state represents an equivalence class. Since equational completion preserves $\mathcal{R}/E$-coherent tree automata (Lemma 106 and Theorem 97), this

property is preserved during completion. During completion of $\mathcal{A}$, the fact that $\forall t \in \mathcal{T}(\mathcal{F}) : \forall q_1, q_2 \in \mathcal{A} : t \rightarrow_{\mathcal{A}}^{\not{\epsilon}*} q_1$ and $t \rightarrow_{\mathcal{A}}^{\not{\epsilon}*} q_2 \implies q_1 = q_2$ can be guaranteed by normalizing the new transitions added to $\mathcal{A}$ using the existing transitions of $\mathcal{A}$. Besides to this, on $\mathcal{R}/E$-coherent tree automata, equational completion computes a subset of E-equivalent $\mathcal{R}$-reachable terms (Theorem 110). From this last result and the fact that there are only a finite number of E equivalent $\mathcal{R}$ reachable terms we get that completion produces a tree automaton recognizing a finite set of E-equivalent terms. Finally, since we know that the produced tree automaton is $\mathcal{R}/E$-coherent and such that $\forall t \in \mathcal{T}(\mathcal{F}) : \forall q_1, q_2 \in \mathcal{A} : t \rightarrow_{\mathcal{A}}^{\not{\epsilon}*} q_1$ and $t \rightarrow_{\mathcal{A}}^{\not{\epsilon}*} q_2 \implies q_1 = q_2$, there is at most one state by E-equivalence class of $\mathcal{R}$-reachable terms. This is enough to prove that the set of states on the completed tree automaton is necessarily finite. □

In section 4.6.4, we show that the verification of safety properties on examples from (Clavel et al., 2008; Meseguer et al., 2003) carried out using equational abstraction can be done using equational completion and the Timbuk tool.

3.3.4 Regular and abstract regular tree model-checking

In this domain, programs or systems are modeled using tree transducers. Since TRS and tree transducers are rather different this makes the comparison difficult. Linear tree transducers can be seen as the particular **IOS-LTS** class of TRSs as shown in Section 2.1.1. This draws strong restriction of the expressive power of tree transducer model. For instance in Section 2.3.2, we have seen that a functional programs can be straightforwardly translated into a TRS. This is not the case for tree transducers. But, because of this restriction, tree transducers enjoy a desirable property w.r.t. to reachability that TRS do not have. As seen in Section 2.3.1, given a tree automaton $\mathcal{A}$ and a linear tree transducer R_T, it is possible to construct a tree automaton $\mathcal{B}$ such that $\mathcal{L}(\mathcal{B}) = R_T(\mathcal{L}(\mathcal{A}))$. This could be seen as "one step of application" of the tree transducer R_T. Let $\mathcal{R}$ be a TRS, $\mathcal{S}$ a set of terms and $\mathcal{R}(\mathcal{S})$ the set defined by $\mathcal{R}(\mathcal{S}) = \{t \mid s \in \mathcal{S} \text{ and } s \rightarrow_{\mathcal{R}} t\}$. In general building a tree automaton recognizing $\mathcal{R}(\mathcal{L}(\mathcal{A}))$ is not easy. In particular, completion is not able to do that, as we show on the next example.

Example 116 *Let $\mathcal{R} = f(x) \rightarrow g(x)$ and $\mathcal{A} = \langle \mathcal{F}, \mathcal{Q}, \mathcal{Q}_f, \Delta \rangle$ the tree automaton such that $\mathcal{Q}_f = \{q_0\}$ and $\Delta = \{f(q_0) \rightarrow q_0, a \rightarrow q_0\}$. We have $\mathcal{L}(\mathcal{A}) = \{f^*(a)\}$ and $\mathcal{R}(\mathcal{L}(\mathcal{A})) = \{f^*(g(f^*(a)))\}$. However, if we apply completion on $\mathcal{R}$ and $\mathcal{A}$, after the first step we obtain $\mathcal{A}^1_{\mathcal{R}} = \mathcal{A} \cup \{g(q_0) \rightarrow q_0\}$. This tree automaton is such that $\mathcal{L}(\mathcal{A}^1_{\mathcal{R}}) = \{(fg)^*(a)\} = \mathcal{R}^*(\mathcal{L}(\mathcal{A}))$, hence $\mathcal{L}(\mathcal{A}^1_{\mathcal{R}}) \supset \mathcal{R}(\mathcal{L}(\mathcal{A}))$.*

With tree transducers, the execution of a program is modeled using repeated application of one (or several) tree transducers. Thus, the objective is to compute or to over-approximate the set $R_T^*(\mathcal{L}(\mathcal{A}))$. There are examples, like the **WOS** class (see Section 2.1.1), where the repeated application of the tree transducers correspond to the application of a single TRS. In that case, we have shown that the construction of the set of reachable terms can be done using completion (see Section 3.3.1). However, in general, such correspondence between tree transducers and TRS is uneasy to define and may also be irrelevant.

Besides to this, when the exact construction of $R_T^*(\mathcal{L}(\mathcal{A}))$ is not possible abstract regular *tree* model-checking benefits of many abstraction techniques inherited of the abstract

regular (word) model-checking community. In Section 2.3.1, we have recalled some of those techniques, taken from (Bouajjani et al., 2006a). Those techniques are based on the definition of an equivalence relation between states of the automaton. Our simplification technique based on equations is, of course, closely related to this one. Recall that in their framework, two different equivalence relations are considered.

- two states are equivalent if their recognized languages are equal for terms of height lesser or equal to a given n;
- given a set of tree automata $P = \{P_1, \ldots, P_N\}$, two states q_1 and q_2 of $\mathcal{A}$ are equivalent if $\{P_i \in P \mid \mathcal{L}(P_i) \cap \mathcal{L}(\mathcal{A}, q_1) \neq \emptyset\}$ and $\{P_i \in P \mid \mathcal{L}(P_i) \cap \mathcal{L}(\mathcal{A}, q_2) \neq \emptyset\}$ are equal.

Those two equivalence relations are incomparable with the one we consider, defined using equations on terms. In other words, none can be defined with the other. The second relation can be used to encode ground equations or equations where left and right hand sides do not share variables. For instance, an equation $f(x) = g(y)$ can be encoded by two tree automata P_1 recognizing the set $\{f(t) \mid t \in \mathcal{T}(\mathcal{F})\}$ and P_2 recognizing the set $\{g(t) \mid t \in \mathcal{T}(\mathcal{F})\}$. However, an equation of the form $f(x) = g(x)$ is not possible to define using this simple encoding. On the opposite, assume that $P = \{P_1, P_2\}$, $\mathcal{L}(P_1) = \{a\}$ and $\mathcal{L}(P_2) = \{b\}$. This set P defines 4 equivalence classes on:

1. states recognizing a and b ($\{P_1, P_2\}$),
2. states recognizing a and not b ($\{P_1\}$),
3. states recognizing b and not a ($\{P_2\}$),
4. states recognizing neither a nor b ($\{\}$).

Classes 2. and 3. can be defined using the equations $a = a$ and $b = b$. However, classes 1. and 4. cannot be defined using simplification by equations as defined in Section 3.2.1. A great advantage offered by the above equivalence relations is that they can be automatically refined if necessary (Bouajjani et al., 2006a). This is not the case for equational completion, but this is ongoing work. Furthermore, refinement of approximations built by standard completion is possible (Boichut et al., 2008).

3.3.5 Static Analysis

In Section 2.3.2, we have seen that approximation of reachable terms have been used for the flow analysis of imperative, functional and logic programs. As explained in that section, since all those analysis share a common mechanism, we focus on the analysis of (Jones, 1987; Jones and Andersen, 2007) because it is defined through TRS and thus makes comparison more accurate. Now, we show how to use equational completion to perform a similar analysis. If necessary, the precision of the approximation can even be easily improved. We show in particular how we can lift the precision of an approximation from a basic flow analysis to a shape analysis. Instead of building by hand the automata produced

by equational completion, we use the Timbuk tool which will be presented more in detail in Chapter 4.

The contributions of (Jones and Andersen, 2007) are to deal with higher order functions and lazy evaluation. Since the higher order part can be done by reusing their encoding of higher-order function into first order TRS (detailed in Section 3.3.5), we here focus on the lazy evaluation part. This example is interesting because the resulting grammar is fully detailed in their paper so that we can compare. In (Jones and Andersen, 2007), the functional program is directly given in its TRS form:

```
g(N) -> first(N, sequence(nil))
first(nil, Xs) -> nil
first(cons(one,M), cons(X,Xs)) -> cons(X,first(M,Xs))
sequence(Y) -> cons(Y,sequence(cons(one,Y)))
```

For any list `N` composed of n `one` symbols, the function g computes the list of the n first elements of the infinite list $[nil, [one], [one, one], \ldots]$. Note that this program needs a lazy or outermost evaluation strategy to terminate because the `sequence` function does not terminate and builds the infinite list $[nil, [one], [one, one], \ldots]$. The initial set of terms is defined by the following automaton:

```
Automaton A0
States q0 ql qa q1 qnil
Final States q0
Transitions
g(ql) -> q0
cons(qa,ql) -> ql
cons(q1,ql) -> ql
cons(q1,qnil) -> ql
cons(qa,qnil) -> ql
nil -> qnil
atom -> qa
one -> q1
```

recognizing all terms of the form `g(l)` where `l` is any list of atoms that can be `one` or any other atom, as in (Jones and Andersen, 2007). Though (Jones and Andersen, 2007) do not comment on it, since the set of atoms is potentially infinite and grammars or automata can only be finite, it is necessary to finitely abstract it. We do this using two distinct constants `one` representing itself and `atom` representing all the other atoms distinct from `one`. In (Jones and Andersen, 2007), the objective is to infer the term structure of possible values for parameters and results of every function f without a priori knowledge on the inputs of the f function. Since completion covers *all* reachable terms, it covers also those that can be reached by a lazy evaluation. In fact, we can achieve exactly the same flow analysis and obtain the same result using equations defining a similar independent attribute approximation. This can be done using contextual equations (see Section 4.1.2). Recall that the intuition behind an independent attribute approximation for a function f is simply to merge together all possible call values for f. Hence, for the function `first` which

has two parameters, such an approximation can be defined using the single contextual equation for `first`: `[first(X,Y), first(Z,U)] => [X=Z  Y=U]`. Similarly, for `sequence` the equation will be: `[sequence(X), sequence(Y)] => [X=Y]`. Using those equations, we obtain a completed automaton of 11 states and 18 transitions. Among the transitions, we can find the following subset recognizing the set of results of the `g` calls:

```
nil -> q13
cons(q10,q13) -> q13
nil -> q10
cons(q3,q10) -> q10
one -> q3
```

which is the same result as the one obtained by (Jones and Andersen, 2007), i.e. any list whose elements are flat lists of `one` symbols.

Adapting the approximation to the property to prove

In order to illustrate the impact of equations on the precision of the approximation, we aim at proving some property on the *reverse* function. This function is classically defined by:

```
append(nil,X) -> X
append(cons(X,Y), Z) -> cons(X, append(Y,Z))
rev(nil) -> nil
rev(cons(X,Y)) -> append(rev(Y), cons(X,nil))
```

Assume that we want to know what can be the result of $rev(l)$ where l can be any flat list of a, b, c and d (in that order) and such that l contains at least one occurrence of each symbol. The language $rev(l)$ is defined by the following tree automaton:

```
Automaton A0
States q0 qla qlb qlc qld qnil qf qa qb qc qd
Final States q0
Transitions
f(qla) -> q0
cons(qa, qla) -> qla
cons(qa, qlb) -> qla
cons(qb, qlb) -> qlb
cons(qb, qlc) -> qlb
cons(qc, qlc) -> qlc
cons(qc, qld) -> qlc
cons(qd, qld) -> qld
cons(qd, qnil) -> qld
nil -> qnil
a -> qa
b -> qb
c -> qc
d -> qd
```

The expected result is, of course, the language of flat list where symbols are in the opposite order and occur at least once. This can be seen as a shape analysis. If we use an independent attribute approximation, as in the previous section, using the following equations:

```
[append(X,Y), append(Z, U)] => [X=Z Y=U]
[rev(X), rev(Y)] => [X=Y]
```

the Timbuk tool produces a tree automaton where state `q29` recognizes the result of $rev(l)$:

```
nil -> q29
cons(q7,q29) -> q29
cons(q8,q29) -> q29
cons(q9,q29) -> q29
cons(q10,q29) -> q29
d -> q10
c -> q9
b -> q8
a -> q7
```

This language is the language of flat lists possibly containing symbols a, b, c and d but in any order. This result is coherent with an independent attribute approximation since all call values of $append$ are merged together. For all function $f(x, y)$, we can improve the approximation by merging together the calling values for x on one side and for y on the other side, only if the calling context are similar. This is the same idea that is used to improve a 0-CFA analysis into a 1-CFA analysis: take the direct calling context into account. For the $append$ symbol, for instance, this can be done using the following kind of equations:

```
[cons(append(X,Y),_), cons(append(Z,U),_)] => [X=Z Y=U]
[cons(_,append(X,Y)), cons(_,append(Z,U))] => [X=Z Y=U]
[append(append(X,Y),_), append(append(Z,U),_)] => [X=Z Y=U]
[append(_,append(X,Y)), append(_,append(Z,U))] => [X=Z Y=U]
```

where we merge call values of $append$ only if the calling context at depth 1 is the same. Even if it improves the precision of the approximation, the resulting automaton still does not preserve the order of symbols in the list. In fact, even by distinguishing between any calling context of depth $k \in \mathbb{N}$ (like in a k-CFA analysis), the approximation would not be precise enough to obtain the result we expect. However, we can construct a different approximation using the single equation:

```
append(append(X,Y),Z)=append(X,Z)
```

Using this equation, we obtain an approximation preserving the order of symbols: the resulting language contains any flat list of d, c, b and a in that order. However, the approximation is still too coarse since there is no guarantee on the occurrence of every symbol in the list. This is due to the fact that, using the previous equation, we have

in particular the following equality: $append(append(cons(b, nil), cons(a, nil)), nil) = append(cons(b, nil), nil)$ meaning that every occurrence of the first term is equivalent to the second. This equation preserve the order of symbols but not their occurrence in the list: the symbol a has disappeared. Finally, it is possible to use the following equations:

```
cons(a, cons(a, X))=cons(a,X)
cons(b, cons(b, X))=cons(b,X)
cons(c, cons(c, X))=cons(c,X)
cons(d, cons(d, X))=cons(d,X)
```

expressing more precisely where contractions of infinite lists have to be performed. These equations permit to construct a completed tree automaton whose recognized language is the expected one, and contains 19 states and 59 transitions.

Dealing with higher-order functions

In (Jones and Andersen, 2007) another contribution is to deal with higher order functions encoding them in a curried way into term rewriting systems. First, here is the functional program.

$cons = \lambda X \lambda Y.(X : Y)$
$double = \lambda X.(X : X)$
$map = \lambda F \lambda L.if\ L = nil\ then\ nil\ else\ (F\ X) : (map\ F\ Xs)$
$f = \lambda X.(map\ double\ X) : (map\ (cons\ a)\ X)$

where 'a' is an atom, ':' and 'nil' are the usual constructors for lists. Now here is its encoding into a TRS, using the curried form, where `app` stands for the application, `fcons` stands for '$cons$' and `cs` stands for ':'

```
app(app(fcons, X), Y) -> cs(X, Y)
app(double, X) -> cs(X, X)
app(app(map, F), nil) -> nil
app(app(map, F), cs(X, Xs)) -> cs(app(F,X), app(app(map, F), Xs))
app(f,X) -> cs(app(app(map, double), X), app(app(map, app(fcons, a)), X))
```

In (Jones and Andersen, 2007), the objective is to infer the term structure of possible values for parameters and results of every function f without a priori knowledge on the inputs of the f function. We can do a similar analysis using Timbuk with the following set of contextual equations:

```
[app(app(map, X), Y), app(app(map, Z), U)] => [X=Z Y=U]
[app(double, X), app(double,Y)] => [X=Y]
```

Note that equations on `double` and `fcons` are even not necessary for the completion to terminate.

Chapter 4

Practical contributions

Experimenting with tree automata and completion, quickly requires a tool to help. The main reason is that, contrary to a word automaton, a tree automaton is a formal structure which is uneasy to draw. Hence, reasoning on tree automata on the paper is hard. The other reason is that, even on simple examples, tree automata completion can produce complex and large automata that cannot be managed by hand.

4.1 The Timbuk library

As a result, we developed several softwares in order to deal with tree automata, all gathered in a library called Timbuk. These software are the Timbuk completion tool (see Section 4.1.2), Taml a command line tool for basic calculations on tree automata (see Section 4.1.3) and Tabi a graphical interface for browsing tree automata (see Section 4.1.4). The Timbuk library is based on the Objective Caml language (Leroy et al., 2000). Timbuk's development started in 2000 and still continues. In particular, the matching algorithm was optimized several times (Section 4.2) and was implemented in the Timbuk completion tool (Section 4.3). The Timbuk tool was also refined in order to deal with non left-linear TRS and conditional TRS (Section 4.4). Recently, a certified checker has been developed in Coq so as to certify the results obtained by completion tools (see Section 4.5). Finally, the library is distributed under the LGPL license since the beginning and has some well identified users, as we will see in Section 4.1.5

4.1.1 History

Before detailing the tools contained in Timbuk library, we present a brief history of the library versions with their contributors.

Timbuk **1.0:** This version contains implementations of tree automata completion as well as all the basic operations on tree automata:

- boolean operations: intersection, union, inversion

- emptiness decision, inclusion decision
- cleaning, renaming
- determinization
- matching of terms over tree automata
- construction of the automaton recognizing $IRR(\mathcal{R})$ for a left-linear $\mathcal{R}$
- normalization of transitions
- parsing, pretty printing
- reading and writing automata to files

However, since the aim was to rapidly experiment with the verification on programs modeled using TRS, optimization efforts were essentially made on the necessary algorithms: completion, intersection, cleaning and emptiness decision. Most of the other available operations are essentially a straightforward implementation of textbook algorithms (Comon et al., 2008). With regards to the approximation technique, Timbuk 1.0 was only proposing interactive approximations, i.e the user was asked to give the state names used to normalize transitions obtained by completion.

Timbuk **2.0** This was improved in Timbuk 2.0 with the introduction of normalization rules (see Section 3.1.2). The two tools Taml and Tabi were also added to the library. Tabi has been developed with a group of Master's students, namely: Boinet Matthieu, Brouard Robert, Cudennec Loic, Durieux David, Gandia Sebastien, Gillet David, Halna Frederic, Le Gall Gilles, Le Nay Judicael, Le Roux Luka, Mallah Mohamad-Tarek, Marchais Sebastien, Martin Morgane, Minier François and Stute Mathieu. In this version, the Timbuk completion tool has been developed in collaboration with Valérie Viet Triem Tong.

The objective of normalization rules is to let Timbuk users state their approximation in a declarative way. Normalization rules revealed to be a very powerful and adaptable way to define approximations. Our experience while Modeling and verifying the SmartRight cryptographic protocol (Genet et al., 2003) (see Section 5.1.3) using Timbuk shows that having declarative approximations is a strength for this kind of verification technique. With normalization rules it is possible to tune the approximation for a specific program or property to prove. Another improvement was the integration in Timbuk of the optimized matching algorithm described in Section 4.2. This algorithm expresses the matching problem as a product between the completed automaton and a tree automaton representing all the left-hand sides of rules. By computing a single product, it is possible to build in one pass all the substitutions corresponding to the application of all the rules at every position in the completed tree automaton. This optimization improved the overall efficiency of Timbuk by a factor 10 on combinatorial TRSs such as cryptographic protocol specifications. This result can be explained by the fact that, with TRSs encoding cryptographic protocol, for each completion step many rewrite rules (in particular those encoding the intruder behavior) can be applied at many positions. The new matching algorithm is clearly well adapted for this kind of TRSs.

Although normalization rules are expressive and efficient, they are not easy to write. As a solution to this problem, Y. Boichut, Pierre-Cyrille Héam and Olga Kouchnarenko proposed to automatically generate those normalization rules from a careful inspection of the TRS. This approach was successful when the TRS encodes a cryptographic protocol and approximations are used to prove secrecy on an unbounded number of interleaved protocol sessions (Boichut et al., 2004). These theoretical results were implemented in the TA4SP verification tool (Boichut, 2005) that is part of the well known AVISPA protocol verification tool (Armando et al., 2005).

Timbuk **3.0** Another way to ease the definition of approximations is to use a simpler language. Normalization rules are based on tree automata formalism and, thus, it is impossible to define such a rule without being a tree automata expert. In Timbuk-3.0 we replaced normalization rules by equations that do not require specific tree automata knowledge to be defined. Although the modification seems of secondary importance, it implied many deep transformations of the completion algorithm (see Section 3.2.3). Thus, the implementation of Timbuk 3.0 is totally new and fully dedicated to reachability analysis. As a consequence, it does no longer provide basic tree automata operations.

Timbuk 2.2 is around 13000 lines of Ocaml and the new version 3.0 re-developed from scratch is around 11000 lines of code. Timbuk 2.2 contains the following tools: Timbuk reachability analysis tool, Taml and Tabi, we now detail.

4.1.2 Timbuk tree automata completion and reachability analysis tool

Basic features: completing and checking (un)reachability

Initially, the library was implemented so as to experiment with the tree automata completion algorithm. The objective was to show that tree automata completion and reachability analysis of TRSs could be an alternative technique for program verification. The Timbuk tool computes exactly or approximately $\mathcal{R}^*(E)$ for given TRS $\mathcal{R}$ and set of initial term E and also check that $\mathcal{R}^*\mathcal{R}(E) \cap Bad = \emptyset$. Sets of terms E, Bad and term rewriting system $\mathcal{R}$ are defined in a Timbuk *specification file*. Let us begin by a simple example with the `basic.txt` specification file where one TRS `R` and three sets of terms `init, check1` and `check2` are defined:

```
(* basic.txt *)

Ops f:1 g:1 a:0

Vars x

TRS R
    f(x) -> g(f(x))


Set init
    f(a)
```

```
Set check1
    g(g(g(g(f(a)))))
    g(f(g(a)))


Automaton check2
States qf q0 q1
Final States qf
Transitions
g(qf) -> qf
f(q1) -> qf
g(q0) -> q1
a -> q0
```

The first line is a comment. The next two declarations are for operators and variables. Then the TRS `R` is defined. The first two sets of terms `init` and `check1` are finite and thus can be defined by extension as lists of ground terms. The last set `check2` is not finite and is defined as a tree automaton. To execute Timbuk on this specification it is enough to execute:

```
timbuk basic.txt
```

on the command line. Then Timbuk reads the above specification and, by default, start a completion on the first TRS and first set of terms. In our case, it tries to compute $\mathtt{R}^*(\mathtt{init})$. When given a finite set of terms using the `Set` constructor, Timbuk transforms it into a tree automaton recognizing exactly this set, i.e. the set $\{\mathtt{f(a)}\}$ in our case. The other sets (and thus other tree automata) associated with names `check1` and `check2` will be used later for verification purpose. Initially Timbuk produces the following output:

```
Completion step: 0
Do you want to:
(c)omplete one step  (use Ctrl-C to interrupt if necessary)
complete (a)ll steps (use Ctrl-C to interrupt if necessary)
(m)erge some states
(s)ee current automaton
(b)rowse current automaton with Tabi
(d)isplay the term rewriting system
(i)ntersection with verif automata
intersection with (o)ther verif automata on disk
search for a (p)attern in the automaton
(v)erify linearity condition on current automaton
(w)rite current automaton, TRS and approximation to disk
(f)orget old completion steps
(e)quation approximation in gamma
(g)amma normalisation rules
(det)erminise current automaton
(u)ndo last step
(q)uit completion
```

```
(c/a/m/s/b/d/i/o/p/v/w/f/e/g/det/u/q)?
```

meaning that the current completion step number is 0 and that Timbuk expect you to give one command. For instance, by typing `s` it is possible to display the current tree automaton being completed:

```
States qterm0:0 qterm1:0

Final States qterm0

Transitions
a -> qterm1
f(qterm1) -> qterm0
```

which is a tree automaton recognizing the set of terms $\{f(a)\}$. Now it is possible to search for reachable terms from $\{f(a)\}$ by doing some completion steps. Type `c` to achieve one completion step. Timbuk finds a new reachable term which corresponds to a new tree automata transition to add to the current tree automaton:

```
Adding transition:

 g(f(qterm1)) -> qterm0
```

Adding this transition to the tree automaton will permit to recognize the term $g(f(a))$ which is reachable from $f(a)$ when applying rule $f(x) \rightarrow g(f(x))$. However the transition `g(f(qterm1)) -> qterm0` has to be normalized first. In our case, the subterm `f(qterm1)` has to be replaced by a state. When the specification does not contain normalization rules, Timbuk interactively asks for normalization instructions. The user can either propose state names or use automatic normalization with new states instead. Automatic normalization causes Timbuk to create a new state `qnew0` to normalize automatically the transition into a set of two normalized transitions equivalent to `g(f(qterm1)) -> qterm0`:

```
Adding transition:

 g(qnew0) -> qterm0

 ... already normalised!


Adding transition:

 f(qterm1) -> qnew0

 ... already normalised!
```

This ends the first completion step. Using the same normalization methodology (i.e. always normalize with new states) it is possible to complete step 2, step 3 and so on, but

completion does not terminate with this strategy. This is not really surprising since rule $f(x) \to g(f(x))$ is not terminating on term $f(a)$ and we are incrementally adding an infinite set of descendants of $f(a)$. However, since this example belongs to the decidable class **L-GFPO** (see Section 2.1.2) and is thus covered by completion (see Section 3.3.1) using the exact normalization strategy (see Definition 113). This can be done by starting again Timbuk in this particular mode:

```
timbuk -strat exact basic.txt
```

Then either type repeatedly `c` or type once `a` for achieving completion until Timbuk succeeds at step 2:

```
Automaton is complete!!
-----------------------
```

Then, it is possible to see the final completed automaton by typing `s`:

```
States qterm0:0 qterm1:0
Final States qterm0
Transitions
a -> qterm1
f(qterm1) -> qterm0
g(qterm0) -> qterm0
```

The obtained automaton recognizes the set of reachable terms. Then, it is possible to check if terms of the sets `check1` and `check2` are `R`-reachable from `f(a)`. This can be done by computing an intersection between $\mathtt{R}^*(\mathtt{init}) = \mathtt{R}^*(\{f(a)\})$ and `check1` and `check2`. Intersections with finite sets or other automata contained in the same specification file can be done by typing `i`, this results in:

```
Intersection with check1 gives (not empty):

States q5:0 q4:0 q3:0 q2:0 q1:0 q0:0

Final States q5

Prior
f(q0) -> q1
a -> q0

Transitions
a -> q0
f(q0) -> q1
g(q1) -> q2
g(q2) -> q3
g(q3) -> q4
g(q4) -> q5
```

for `check1`, meaning that some terms of `check1` (in fact the term $g(g(g(g(f(a))))))$ is reachable from $f(a)$ using `R`. For `check2`, a similar test results in:

```
Intersection with check2 gives (the empty automaton):

States

Final States

Transitions
```

meaning that the term of `check2` are all not reachable by rewriting $f(a)$ with `R`. Another way to check that a term is (un)reachable is to use the pattern detection of this tool. After the completion of this example, it is possible to type `p` to look if a given pattern can be found in the automaton.

```
Type a term and hit Return: g(g(g(x)))

Solutions:
Occurence in state qterm0!
solution 1: x <- qterm0
```

If we look for a term of the form $g(g(g(x)))$ in the completed tree automaton $\mathcal{A}^*_{\mathcal{R}}$, the answer is that such a term can be found and both a position (`Occurrence in...`) and a substitution (`solution...`) is given. On this example, this means that $g(g(g(\texttt{qterm0}))) \rightarrow^*_{\mathcal{A}^*_{\mathcal{R}}}$ `qterm0`. In other words, there exists a ground substitution σ and a state `qterm0` such that $g(g(g(x)))\sigma \rightarrow^*_{\mathcal{A}^*_{\mathcal{R}}}$ tt qterm0, i.e. $g(g(g(x)))\sigma$ is recognized by a state in $\mathcal{A}^*_{\mathcal{R}}$. This means that there exists a reachable term with subterm $g(g(g(x)))\sigma$. Moreover, if the occurrence state is final then the term belongs to $\mathcal{L}(\mathcal{A}^*_{\mathcal{R}})$ and thus the term is reachable. Note that it is the case here. If such occurrence and state are not found the subterm is not reachable:

```
Type a term and hit Return: f(g(g(a)))

Pattern not found!
```

Defining approximations using normalization rules

This is what can be done on TRS and initial sets of terms for which completion *naturally* terminates, i.e. problems having a finite set of reachable terms or TRSs in the decidable classes covered by the completion (see Section 3.3.1). Recall that normalization rules (see Section 3.1.2) in Timbuk, are rules of the form:

$$[s \rightarrow x] \rightarrow [l_1 \rightarrow r_1 \ldots l_n \rightarrow r_n]$$

where s, $l_1, \ldots, l_n$ are terms that may contain symbols, variables and states, and x, r_1, $\ldots$, r_n are either states or variables such that if r_i is a variable then it is equal to x. To normalize a transition of the form $t \rightarrow q'$, we match the pattern s on t and x on q', obtain a substitution σ and then we normalize t with the rewrite system $\{l_1\sigma \rightarrow r_1\sigma, \ldots, l_n\sigma \rightarrow r_n\sigma\}$ where $r_1\sigma, \ldots, r_n\sigma$ are necessarily states. We illustrate the use of normalization rules on the following specification:

```
(* normrules.txt *)

Ops f:1 g:1 a:0

Vars x z

TRS R
    g(x) -> f(g(f(x)))

Set init
    g(a)

Set check2
    f(f(f(g(f(f(a))))))

Approximation Simple
States q0 q1 q2 q3
Rules
      [f(g(f(q0))) -> z] -> [f(q0) -> q2 g(q2) -> q3]
      [f(g(f(q2))) -> z] -> [f(q2) -> q0 g(q0) -> q1]
      [f(g(f(x))) -> z]  -> [f(x) -> q0  g(q0) -> q1]
```

In this example the set $\mathcal{R}^*(\{g(a)\}) = \{f^n(g(f^n(a))) \mid n \in \mathbb{N}\}$ which is not regular and thus cannot be computed exactly using a tree automaton. However, it is possible to over-approximate it. The objective, here, is to prove that the term $f^3(g(f^2(a)))$ is not reachable. In this case, we can use a simple approximation where we collapse chains of $f(f(f(\ldots)))$ such that we only distinguish between even and odd number of f symbols above and below the g symbols. This is done by the above normalization rules whose role is to normalize transitions of the form $f(g(f(q))) \to q'$, i.e. determine state names for subterms $f(q)$ and $g(f(q))$. For that purpose, we use distinct states q_0, q_1, q_2, q_3 such that:

- q_0 recognizes an odd number of f below g
- q_1 recognizes a g above an odd number of f
- q_2 recognizes an f above a q_0, i.e. an even number of f below g
- q_3 recognizes a g above an even number of f

Starting Timbuk completion on this specification results in the following tree automaton:

```
States q0:0 q1:0 q2:0 q3:0 qterm0:0 qterm1:0

Final States qterm0

Transitions
a -> qterm1
g(qterm1) -> qterm0
f(qterm1) -> q0
```

```
g(q0) -> q1
f(q1) -> qterm0
f(q0) -> q2
g(q2) -> q3
f(q3) -> q1
f(q2) -> q0
f(q1) -> q3
```

which recognizes the language $\{(ff)^*(g((ff)^*(a))), f((ff)^*(g((ff)^*(f(a)))))\}$, i.e. terms of the form $f^n(g(f^m(a)))$ such that n and m are both even or both odd. Thus the intersection with $\{f^3(g(f^2(a)))\}$ is empty. Note that, to ease the understanding of the language recognized by an initial or completed tree automaton we can use the graphical tool Tabi which is presented on that example in Section 4.1.4.

Defining approximations using equations, conditional equations and contextual equations

A similar approximation and result can be obtained using Timbuk 3.0 and equational approximation. The syntax is very similar to the one used in Timbuk 2.0 except the field `Patterns` that defines a list of term of $\mathcal{T}(\mathcal{F}, \mathcal{X})$ that have to be matched on the terms recognized by the completed automaton after each step of completion. If a solution is found then the term is recognized by the completed automaton. If the pattern has to be matched everywhere in the tree automaton and not only at top position of terms, we can use the `SubPatterns` flag instead.

```
% equations.txt

Ops f:1 g:1 a:0

Vars x y

TRS R
    g(x) -> f(g(f(x)))

Set init
    g(a)

Patterns
     f(f(f(g(f(f(a))))))

Equations Simple
Rules
     f(f(x))=x
```

Note that the approximation definition corresponds exactly to what has been proposed above using normalization rules but in a simpler and more concise way. Interactions with Timbuk 3.0 are also simpler than with Timbuk 2.0. On the previous specification, we can run the completion by simply typing:

```
timbuk 8 equations.txt
```

where 8 is the maximal number of completion steps that are going to be performed. On the previous specification, Timbuk 3.0 ends as follows:

```
States q1 q2 q3 q4 q5 q6 q7
Final States q1
Transitions
f(q6) -> q7
g(q5) -> q6
f(q3) -> q6
a -> q5
f(q2) -> q5
f(q3) -> q4
g(q2) -> q3
f(q6) -> q3
f(q5) -> q2
g(q5) -> q1
f(q3) -> q1

Step: 4

Automaton complete!
------------------
```

meaning that the tree automaton has successfully been completed in 4 steps and that the pattern has not been found, thus proving that the term $f^3(g(f^2(a)))$ is not reachable. The above example shows that equations are far more declarative than normalization rules to define approximations. However, normalization rules have some facilities that have disappeared in equations: the ability to define contextual approximations. With normalization rules it is possible to state that the approximation has to be carried out under a particular context. For instance, the following normalization rule:

$$[f(g(x)) \to z] \to [g(x) \to q]$$

states that any term of the form $g(x)$ is normalized by a unique state q but only if it appears under an f. A similar approximation, where all terms of the form $g(x)$ have to be merged together only if they appear under an f, is impossible to achieve using equations. However, such contextual approximations revealed to be crucial for static analysis: the independent attribute approximation of Section 3.3.5 as well as for other analysis on more realistic programs (see Section 5.2). This is the reason why, in Timbuk 3.0 we added *contextual equations* that can have three different forms:

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

Assuming that $\mathcal{A}$ is a tree automaton, applying those contextual equations on $\mathcal{A}$ is done as follows. The right-hand side of $\Rightarrow$ is a list of equation to be applied on $\mathcal{A}$ provided that the left-hand side can be matched on $\mathcal{A}$. A left-hand side of the form $[s]$ means that we look only for matching solution for s on $\mathcal{A}$, for $[s, t]$ we look for solutions for both s and t independently and for $[s = t]$ we look for solutions such that matches for s and t belong to the same equivalence class, i.e. recognized by the same state of $\mathcal{A}$. Going back to our approximation example, where we wanted to merge all the terms of the form $g(x)$ provided that they appear under an f, it can be defined by the following contextual equation:

$$[f(g(x)), f(g(y))] \Rightarrow [g(x) = g(y)]$$

Finally Timbuk 3.0 also offer a restricted form of *conditional equations*. These conditional equations are of the form:

$$f(g(x), y) = f(h(x), z) \text{ if } x \neq y \wedge y \neq z$$

Simplification (see Section 3.2.1) is performed using this equation if and only if, when matching $f(g(x), y)$ and $f(h(x), z)$ in the automaton, the states associated to x, y and z satisfy the given condition.

4.1.3 Taml

Taml is a very simple interactive tool for achieving basic tree automata computations. The aim of this section is essentially to show what a Taml working session looks like and how Taml functions and Ocaml top-level nicely interact. We use Taml to check simple properties on tree automata like inclusion, emptiness, ..., as well as compute intersections or the automaton recognizing $IRR(\mathcal{R})$ for a given left-linear TRS $\mathcal{R}$. Actually, Taml is an Ocaml interpreter extended with Timbuk library functionalities and pretty printing (see (Genet, 2003) for reference manual of Taml and see (Leroy et al., 2000) for details about Ocaml syntax). Taml permits to define alphabets, sets of variables, terms, term rewriting systems and tree automata and also to apply basic tree automata operations using specific functions. We define an alphabet `f`, a variable set `v` and a ground term `t` as follows (where the `#` symbol represents the Taml prompt).

```
# let f= alphabet "app:2 cons:2 nil:0 a:0 b:0";;
val f : Taml.Alphabet.t = app:2 cons:2 nil:0 a:0 b:0

# let v= varset "x y z u";;
val v : Taml.Variable_set.t = x y z u

# let t= term f  v "cons(a, cons(b, nil))";;
val t : Taml.Term.t = cons(a,cons(b,nil))
```

Since Taml is built on the top of a complete Ocaml interpreter, it is thus possible to use usual Ocaml syntax facilities and also to combine Taml functions with usual Ocaml functions. For instance, it is possible to define a specific `term` function specialized for alphabet `f` and variable set `v` in the following way:

```
# let fvterm= term f v;;
val fvterm : string -> Taml.Term.t = <fun>
```

Now it is possible to construct a list of terms built on alphabet `f` and variable set `v` using the specialized function `fvterm` as well as Ocaml `List.map` function (mapping a function to every element of a list) in the following way:

```
# let l= List.map fvterm ["app(cons(a, nil),cons(b, cons(b, nil)))"; "a"; "cons(a,nil)
val l : Taml.Term.t list = [app(cons(a,nil),cons(b,cons(b,nil)));a;cons(a,nil)]
```

Similarly we can construct term rewriting systems and tree automata directly in the interpreter.

```
# let tt= trs f v "app(nil, x) -> x   app(cons(x, y), z) -> cons(x, app(y, z))";;
val tt : Taml.Rewrite.t =
  app(nil,x) -> x
  app(cons(x,y),z) -> cons(x,app(y,z))

# let aut= automaton f "
States qa qb qla qlb qf
Final States qf
Transitions
        a -> qa
        b -> qb
        cons(qa, qla) -> qla
        nil -> qla
        cons(qb, qlb) -> qlb
        nil -> qlb
        app(qla,qlb) -> qf";;

val aut : Taml.Automaton.t =
  States qa:0 qb:0 qla:0 qlb:0 qf:0

Final States qf

Transitions
a -> qa
b -> qb
cons(qa,qla) -> qla
nil -> qla
cons(qb,qlb) -> qlb
nil -> qlb
app(qla,qlb) -> qf
```

We can prove that a given term is recognized by a particular state in a tree automaton.

```
# let t1= List.hd l;;
val t1 : Taml.Term.t = app(cons(a,nil),cons(b,cons(b,nil)))

# let s= state "qf";;
```

```
val s : Taml.Automaton.state = qf

# run t1 s aut;;
- : bool = true
```

We can compute the automaton recognizing the set of terms irreducible by TRS `tt` by typing the following command:

```
# let aut_irr= irr f tt;;
val aut_iff : Taml.Automaton.t =
  States q2:0 q1:0 q0:0

Final States q0 q1 q2

Transitions
b -> q2
a -> q2
nil -> q1
app(q2,q2) -> q2
app(q2,q1) -> q2
cons(q1,q1) -> q0
cons(q2,q2) -> q0
cons(q2,q1) -> q0
cons(q1,q2) -> q0
cons(q0,q0) -> q0
cons(q2,q0) -> q0
cons(q1,q0) -> q0
cons(q0,q2) -> q0
cons(q0,q1) -> q0
app(q2,q0) -> q2
```

Now, we can check that no term of `aut` is irreducible by the following computation:

```
let intersec= simplify (inter aut aut_irr);;
val intersec : Taml.Automaton.t = States

Final States

Transitions
```

where `inter` is the tree automata intersection and `simplify` combines two cleanings of the tree automaton: an accessibility and an utility cleaning. Accessibility cleaning of a tree automaton $\mathcal{A}$ suppress all states q (and corresponding transitions) such that $\forall t \in \mathcal{T}(\mathcal{F}) : t \not\rightarrow_{\mathcal{A}}{}^{*} q$. In the tree automaton $\mathcal{A}$, utility cleaning suppresses every state q such that $\forall t \in \mathcal{T}(\mathcal{F}) : \forall q' \in \mathcal{Q}_f$ and for all derivation $t \rightarrow_{\mathcal{A}}{}^{*} q'$, q does not appear in the derivation. The above results proves that no term recognized by `aut` is irreducible. On the opposite, we can check that *all* the terms of `aut` are reducible terms as follows.

```
# let inv= inverse aut_irr;;
val inv : Taml.Automaton.t =
  States q0:0 q1:0 q2:0 q3:0 q4:0 q5:0

Final States q5

Transitions

nil -> q3
a -> q4
b -> q4
app(q0,q0) -> q5
app(q0,q4) -> q5
app(q0,q5) -> q5
app(q3,q0) -> q5
app(q5,q0) -> q5
app(q3,q5) -> q5
app(q0,q3) -> q5
app(q2,q3) -> q2
app(q4,q0) -> q4
app(q4,q3) -> q4
app(q4,q4) -> q4
app(q2,q4) -> q2
app(q2,q2) -> q2
app(q4,q2) -> q2
app(q2,q0) -> q2
app(q2,q1) -> q2
app(q3,q3) -> q5
app(q3,q4) -> q5
app(q4,q1) -> q2
app(q4,q5) -> q5
app(q5,q5) -> q5
app(q5,q3) -> q5
app(q5,q4) -> q5
cons(q3,q2) -> q1
cons(q3,q0) -> q0
cons(q3,q1) -> q1
cons(q1,q3) -> q1
cons(q0,q4) -> q0
cons(q5,q5) -> q5
cons(q3,q3) -> q0
cons(q3,q4) -> q0
cons(q2,q1) -> q1
cons(q4,q3) -> q0
cons(q4,q5) -> q5
cons(q2,q4) -> q1
cons(q3,q5) -> q5
cons(q4,q2) -> q1
cons(q2,q2) -> q1
cons(q5,q0) -> q5
cons(q1,q2) -> q1
cons(q1,q0) -> q1
cons(q1,q1) -> q1
cons(q0,q3) -> q0
cons(q4,q4) -> q0
cons(q0,q2) -> q1
cons(q0,q1) -> q1
cons(q2,q0) -> q1
cons(q0,q5) -> q5
cons(q0,q0) -> q0
cons(q4,q1) -> q1
cons(q4,q0) -> q0
cons(q1,q4) -> q1
cons(q2,q3) -> q1
cons(q5,q3) -> q5
cons(q5,q4) -> q5

# is_included aut inv;;
- : bool = true
```

where `is_included a1 a2` tests that the language recognized by automaton `a1` is included in the language recognized by `a2`. The `inverse` operation implements the usual complement operation on languages. The above results guarantees that all terms of `aut` are reducible.

4.1.4 Tabi

Tabi has been developed in collaboration with a group of Master's students: Boinet Matthieu, Brouard Robert, Cudennec Loic, Durieux David, Gandia Sebastien, Gillet David, Halna Frederic, Le Gall Gilles, Le Nay Judicael, Le Roux Luka, Mallah Mohamad-Tarek, Marchais Sebastien, Martin Morgane, Minier François and Stute Mathieu. This tool offers a graphical interface designed to browse automata and build representatives of their recognized language. Let us consider again the tree automaton of Section 4.1.2:

```
States q0:0 q1:0 q2:0 q3:0 qterm0:0 qterm1:0

Final States qterm0
```

```
Transitions
a -> qterm1
g(qterm1) -> qterm0
f(qterm1) -> q0
g(q0) -> q1
f(q1) -> qterm0
f(q0) -> q2
g(q2) -> q3
f(q3) -> q1
f(q2) -> q0
f(q1) -> q3
```

Above, we stated that this tree automaton recognizes the language $\{(ff)^*(g((ff)^*(a))), f((ff)^*(g((ff)^*(f(a)))))\}$, i.e. every term is of the form $f^n(g(f^m(a)))$ such that n and m are both even or both odd. This can be inferred from a careful browsing of this tree automaton. Using Tabi it is possible to interactively build terms recognized by states of the tree automaton. Starting from the final state (here `qterm0`) and by clicking on this state it is possible to chose in a list box which transition to unfold, between `g(term1) -> qterm0` and `f(q1) -> qterm0`. Assume that we choose the second one, we thus have chosen the top symbol, i.e. `f` and can continue the unfolding on `q1`. After several unfolding, we reach the situation depicted in Figure 4.1. In this figure, we can remark that when the mouse pointer is over a subterm, here `f(q3)`, then it is highlighted in green and its recognizing state, here `q1`, is shown as a label.

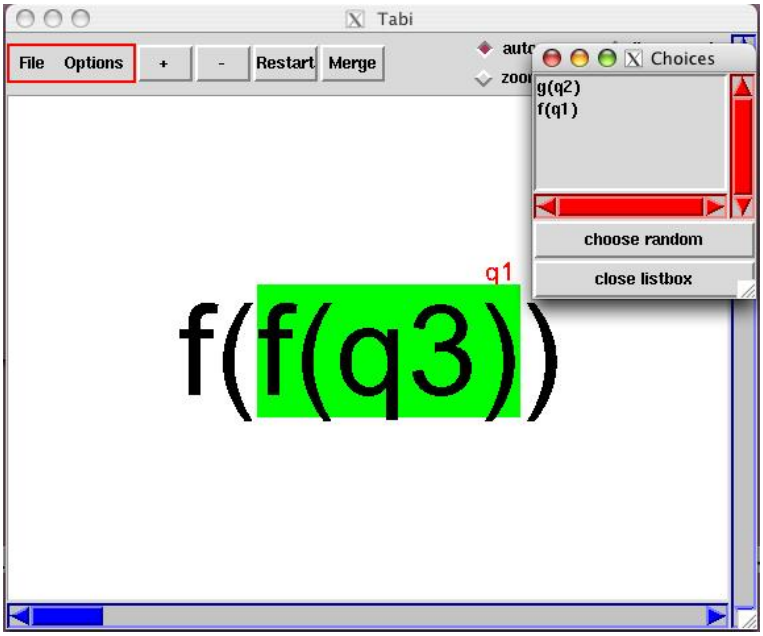


Figure 4.1: An example of automata browsing with Tabi

For a given state, instead of building terms by hand, Tabi also proposes to produce representatives randomly. If we do so on the final state `qterm0`, we obtain the screen displayed in Figure 4.2. We can check that the values computed randomly all verify the property given above, i.e. that terms are all of the form $f^n(g(f^m(a)))$ with n and m both even or both odd. Finally, when the terms under study become huge, Tabi also permits to switch from *linear mode* to *tree mode* at the subterm level for a better readability (see Figure 4.3).

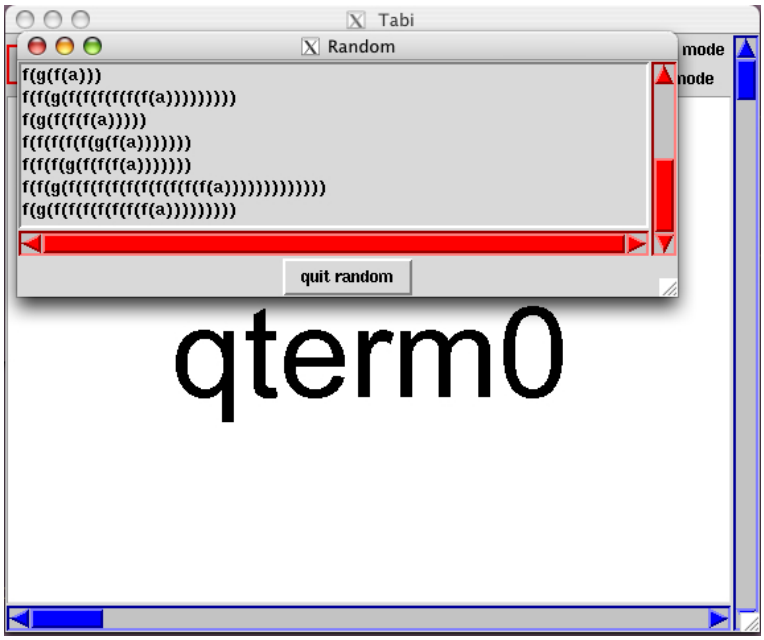


Figure 4.2: Random generation of terms recognized by a given state

4.1.5 Known users

Besides the experiment we performed (see Section 5), some other research group use Timbuk for software verification. Frédéric Oehl and David Sinclair from Dublin University have proposed an hybrid verification approach for cryptographic protocols combining assisted proofs with Isabelle/HOL and approximations with Timbuk (Oehl and Sinclair, 2001; Oehl et al., 2003). Later, Pierre-Cyrille Héam, Yohan Boichut and Olga Kouchnarenko have used Timbuk to develop TA4SP (Boichut et al., 2004; Boichut, 2005), one of the automatic proof back-ends of the AVISPA tool (Armando et al., 2005). AVISPA is a very well know cryptographic protocol verification tool. Among the 4 verification back-ends of AVISPA, TA4SP is the only one to prove secrecy properties for an unbounded number of sessions thanks to Timbuk approximations.

Finally, Timbuk is used as a basic implementation of tree automata in other works and tools. Gaël Patin, Mihaela Sighireanu and Tayssir Touili (Patin et al., 2007a) use Tim-

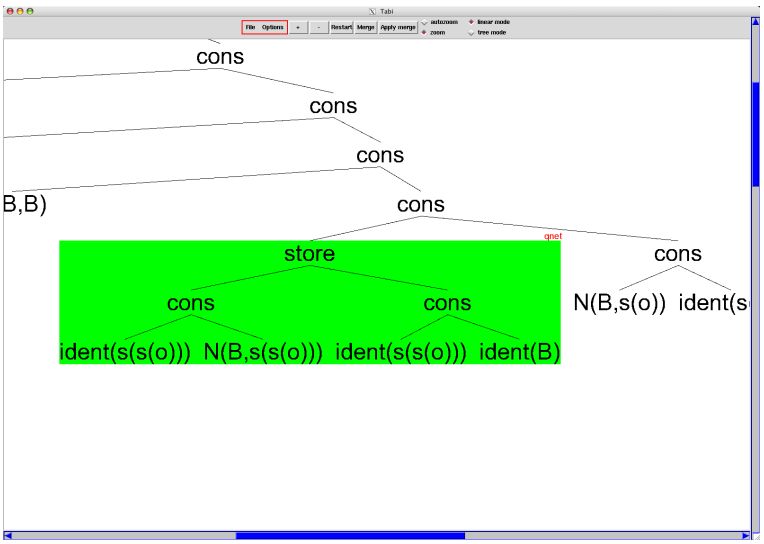


Figure 4.3: Switching from linear to tree display mode on subterms

buk for the analysis of Multi-threaded Dynamic and Recursive Programs and it is part of the related verification tool: SPADE (Patin et al., 2007b). Timbuk is also a part of the PRESS (Sighireanu and Touili, 2006) verification tool used for the verification of process rewrite systems by Mihaela Sighireanu and Tayssir Touili. Finally, Ahmed Bouajjani, Peter Habermehl, Adam Rogalewicz and Tomás Vojnar use Timbuk for experimenting with Abstract Regular Tree Model Checking (Bouajjani et al., 2006b).

4.2 Tree automata matching optimization

After having presented the practical use of Timbuk, we introduce several optimizations and extensions of the tool which have been done. The first optimization concerns the matching operation of Timbuk. The efficiency of the matching algorithm is crucial for the overall efficiency of completion. Given a rewrite rule $l \to r$ and a tree automaton $\mathcal{A}$, matching consists of finding all regular language substitutions σ and states q of $\mathcal{A}$ such that $l\sigma \to_{\mathcal{A}}^{*} q$. In the next section, we give the basic matching algorithm. Then, in Section 4.2.2 we propose an optimized matching algorithm for epsilon free tree automata. The later is extended to the case of tree automata with epsilon transitions in Section 4.2.3. Details on the real implementation of the matching algorithm in Timbuk 3.0 will be given in Section 4.3.

4.2.1 Basic matching algorithm

This algorithm proposed in (Genet, 1997) is close to a standard matching algorithm on terms. It is defined using deduction rules over specific formulas called *matching problems*.

In the following, a *matching problem* is a quantifier-free first order formula built on literals $\bot$, $s \trianglelefteq c$ where $s \in \mathcal{T}(\mathcal{F}, \mathcal{X})$, $c \in \mathcal{T}(\mathcal{F} \cup \mathcal{Q})$, and closed by the connectives $\vee$ and $\wedge$. The literal $s \trianglelefteq c$ should be read as s is matched over c. An empty conjunction $\bigwedge_\emptyset$ is a trivially true matching problem.

Definition 117 *Let ϕ, ϕ_1, ϕ_2 be matching problems, $s \in \mathcal{T}(\mathcal{F}, \mathcal{X})$ be a term, $c \in \mathcal{T}(\mathcal{F} \cup \mathcal{Q})$, and $\mathcal{A} = \langle \mathcal{F}, \mathcal{Q}, \mathcal{Q}_f, \Delta \rangle$ a tree automaton. A* solution *to the matching problem ϕ is a regular language substitution (see Definition 59) $\sigma \in \Sigma(\mathcal{Q}, \mathcal{X})$ such that*

- *if $\phi = s \trianglelefteq c$, then $s\sigma \rightarrow^*_\Delta c$, or*
- *if $\phi = \phi_1 \wedge \phi_2$, then σ is a solution of ϕ_1 and a solution of ϕ_2, or*
- *if $\phi = \phi_1 \vee \phi_2$, then σ is a solution of ϕ_1 or a solution of ϕ_2.*

$\diamond$

For simplicity, we assume that matching is applied to automata without epsilon-transitions. If necessary, epsilon-transitions can be replaced by sets of epsilon-free transitions (see (Comon et al., 2008)).

Definition 118 *Let $\mathcal{A} = \langle \mathcal{F}, \mathcal{Q}, \mathcal{Q}_f, \Delta \rangle$ be a tree automaton, $f \in \mathcal{F}$, $\mathcal{A}r(f) = n$, $g \in \mathcal{F}$, $\mathcal{A}r(g) = m$, $q, q_1, \ldots, q_n, q'_1, \ldots, q'_n \in \mathcal{Q}$, $c_1, \ldots, c_d \in \mathcal{T}(\mathcal{F} \cup \mathcal{Q})$, $s, s_1, \ldots, s_n \in \mathcal{T}(\mathcal{F}, \mathcal{X})$ and ϕ_1, ϕ_2, ϕ_3 be non-empty matching problems. The matching algorithm consists in normalizing any matching problem of the form $s \trianglelefteq q$ by the following set of rules.*

Decompose
$$\frac{f(s_1, \ldots, s_n) \trianglelefteq f(q_1, \ldots, q_n)}{s_1 \trianglelefteq q_1 \wedge \ldots \wedge s_n \trianglelefteq q_n}$$

Clash
$$\frac{f(s_1, \ldots, s_n) \trianglelefteq g(q'_1, \ldots, q'_m)}{\bot}$$

Configuration
$$\frac{s \trianglelefteq q}{s \trianglelefteq c_1 \vee \ldots \vee s \trianglelefteq c_d \vee \bot}$$

if $s \notin \mathcal{X}$, for all $c_i \in \mathcal{T}(\mathcal{F} \cup \mathcal{Q})$ $i = 1 \ldots d$ such that $c_i \rightarrow q \in \Delta$.

Moreover, after each application of any of these rules, matching problems are normalized by the following set of rules ξ:

$$\frac{\phi_1 \wedge (\phi_2 \vee \phi_3)}{(\phi_1 \wedge \phi_2) \vee (\phi_1 \wedge \phi_3)} \qquad \frac{\phi_1 \vee \bot}{\phi_1} \qquad \frac{\phi_1 \wedge \bot}{\bot}$$

$\diamond$

Correctness, completeness and termination of the algorithm comes from the following theorem of (Genet, 1997).

Theorem 119 *Given $\mathcal{A} = \langle \mathcal{F}, \mathcal{Q}, \mathcal{Q}_f, \Delta$, $s \in \mathcal{T}(\mathcal{F}, \mathcal{X})$, and $q \in \mathcal{Q}$, every matching problem $s \trianglelefteq q$ has a normal form such that*

- *if it is $\perp$ then there is no regular language substitution σ s.t. $s\sigma \rightarrow^*_\Delta q$,*
- *if it is empty, then for all regular language substitution σ, $s\sigma \rightarrow^*_\Delta q$,*
- *otherwise, the normal form is a disjunction $\bigvee_{i=1}^{k} \phi_i$ s.t. $\phi_i = \bigwedge_{j=1}^{n_i} x_j^i \trianglelefteq q_j^i$, where $x_j^i \in \mathcal{X}$ and $q_j^i \in \mathcal{Q}$, and $\sigma_1 = \{x_j^1 \mapsto q_j^1 \mid j = 1 \ldots n_1\}, \ldots, \sigma_k = \{x_j^k \mapsto q_j^k \mid j = 1 \ldots n_k\}$ are the only regular language substitutions s.t. $s\sigma_i \rightarrow^*_\Delta q$.*

Thanks to this algorithm, for a given rule $l \rightarrow r$ and a given state q, it is possible to find every regular language substitution σ s.t. $l\sigma \rightarrow^*_\Delta q$.

Example 120 *Let $\mathcal{A} = \langle \mathcal{F}, \mathcal{Q}, \mathcal{Q}_f, \Delta \rangle$, where $\mathcal{F} = \{f, g, a\}$, $\mathcal{Q} = \{q_0, q_1\}$, $\mathcal{Q}_f = \{q_0\}$ and $\Delta = \{f(q_1) \rightarrow q_0, g(q_1) \rightarrow q_1, a \rightarrow q_1\}$. The language $\mathcal{L}(A) = \{f(g^*(a))\}$. Let $\mathcal{R} = \{f(g(x)) \rightarrow g(f(x))\}$. If we apply matching on $f(g(x)) \trianglelefteq q_0$, we obtain the following deductions, where the name of the applied rule is given on the right, and normalization with simplification rules are omitted:*

$f(g(x)) \trianglelefteq q_0$	*rule* **Configuration**
$f(g(x)) \trianglelefteq f(q_1)$	*rule* **Decompose**
$g(x) \trianglelefteq q_1$	*rule* **Configuration**
$g(x) \trianglelefteq g(q_1) \ \vee \ g(x) \trianglelefteq a$	*rule* **Clash**
$g(x) \trianglelefteq g(q_1)$	*rule* **Decompose**
$x \trianglelefteq q_1$	

Let σ be the regular language substitution $\sigma = \{x \mapsto q_1\}$. Thus, we deduced that $l\sigma = f(g(q_1)) \rightarrow^*_\Delta q_0$.

4.2.2 An optimized algorithm for epsilon-free automata

Given a TRS $\mathcal{R}$ and a tree automaton $\mathcal{A}$, for computing all regular language substitutions, it is necessary to run the previous algorithm for ever rule $l \rightarrow r \in \mathcal{R}$ and every state $q \in \mathcal{Q}$. Now, we propose a more efficient algorithm for epsilon-free tree automata dealing with all rules of $\mathcal{R}$ at the same time. Note that the case of automata with epsilon transition will be treated as an extension of this algorithm in Section 4.2.3. The idea behind the optimized matching algorithm is to represent the set of terms to be matched, i.e. the set of left-hand sides of $\mathcal{R}$ by a tree automaton $\mathcal{A}_\mathcal{R}$ and use the tree automata intersection algorithm between $\mathcal{A}_\mathcal{R}$ and $\mathcal{A}$, the automaton to complete, to compute the set of substitutions. First, for every term $t \in \mathcal{T}(\mathcal{F}, \mathcal{X})$, we define a tree automaton whose language is exactly $\{t\}$ using abstraction and normalization functions defined in Section 3.1.

Definition 121 (Term automaton) *Let S be a finite set $S \subseteq \mathcal{T}(\mathcal{F}, \mathcal{X})$, and consider $\mathcal{S}ub$ the set of all the subterms of S, $\mathcal{Q}_S$ a set of states, and $\alpha : \mathcal{S}ub \rightarrow \mathcal{Q}_t$ an injective abstraction function. The term automaton for S is defined by $\mathcal{A}_{\alpha,S} = \langle \mathcal{F}, \mathcal{Q}_t, \mathcal{Q}_{Sf}, \Delta_S \rangle$ where $\mathcal{Q}_{Sf} = \{top_\alpha(t) \mid t \in S\}$ and $\Delta_S = \bigcup_{t \in S} Norm_\alpha(t \rightarrow top_\alpha(t))$.* $\diamond$

Proposition 122 *If $S \subseteq \mathcal{T}(\mathcal{F}, \mathcal{X})$, α is an injective abstraction and $\mathcal{A}_{\alpha,S}$ its term automaton then*

$$\mathcal{L}(\mathcal{A}_{\alpha,S}) = S$$

PROOF. The proof is done by an inductive reasoning over the depth of terms t of S. □

Let us now define the substitution automaton $\mathcal{A}_\cap = \mathcal{A}_{\alpha,S} \cap \mathcal{A}$ which recognizes a set of $\mathcal{Q}$-instances which corresponds to the substitutions solutions of the matching of terms of S on $\mathcal{A}$.

Definition 123 (Substitution automaton) *Let $\mathcal{A} = \langle \mathcal{F}, \mathcal{Q}, \mathcal{Q}_f, \Delta \rangle$ be an automaton, α an injective abstraction function and $\mathcal{A}_{\alpha,S} = \langle \mathcal{F}, \mathcal{Q}_S, \mathcal{Q}_{St}, \Delta_t \rangle$ the automaton of $S \subseteq \mathcal{T}(\mathcal{F}, \mathcal{X})$. We denote by $\mathcal{A}_\cap = \langle \mathcal{F} \cup (\mathcal{X} \times \mathcal{Q}), \mathcal{Q}_\cap, \mathcal{Q}_{f,\cap}, \Delta_\cap \rangle$ the substitution automaton of S in $\mathcal{A}$ by:*

- $\mathcal{Q}_\cap = \mathcal{Q}_S \times \mathcal{Q}$
- $\mathcal{Q}_{f,\cap} = \mathcal{Q}_{Sf} \times \mathcal{Q}$
- $$\begin{aligned} \Delta_\cap = \quad & \{f((q_1, q'_1), \ldots, (q_n, q'_n)) \to (q_{n+1}, q'_{n+1}) \mid q_i \in \mathcal{Q}_t,\ q'_i \in \mathcal{Q}, \\ & \qquad f(q_1, \ldots, q_n) \to q_{n+1} \in \Delta_S,\ f(q'_1, \ldots, q'_n) \to q'_{n+1} \in \Delta\} \\ & \cup \{(x, q) \to (q_x, q) \mid x \to q_x \in \Delta_S,\ q \in \mathcal{Q}\} \end{aligned}$$

◇

Definition 124 ($\mathcal{Q}$-instance) *Let $\mathcal{F}$ be an alphabet, $\mathcal{Q}$ a set of states, $\mathcal{X}$ a set of variables, we inductively define the set $\mathcal{Q}$-instances:*

- *every pair (x, q) for $x \in \mathcal{X}$ and $q \in \mathcal{Q}$ is a $\mathcal{Q}$-instance*
- *$\forall f \in \mathcal{F}\ ar(f) = n$, if $i_1, \ldots, i_n$ are some $\mathcal{Q}$-instances then $f(i_1, \ldots, i_n)$ is also a $\mathcal{Q}$-instance.*

◇

Definition 125 (Recognized $\mathcal{Q}$-instance) *Let $\mathcal{A}_\cap = \langle \mathcal{F}, \mathcal{Q}_\cap, \mathcal{Q}_{\cap f}, \Delta_\cap \rangle$ be a substitution automaton. $\mathcal{A}_\cap$ defines the set of all $\mathcal{Q}$-instances i such that there exists a final state $q \in \mathcal{Q}_{\cap f}$ verifying $i \to^*_{\Delta_\cap} q$* ◇

Definition 126 (Regular language substitution defined by a $\mathcal{Q}$-instance) *A $\mathcal{Q}$-instance s inductively defines a set of regular language substitutions*

- *if $s = (x, q)$, then s defines the regular language substitution $\{x \mapsto q\}$*
- *if $s = (a, a)$ then s defines the empty substitution*
- *if $s = f(s_1, \ldots, s_n)$, then if $\{\sigma_{i,j}\}_{i \in I_j}$ are the substitutions associated to s_j, s define the set of substitutions $\tau = \sigma_{1,i_1} \circ \ldots \circ \sigma_{n,i_n}$.*

◇

Example 127 *Let $\mathcal{A}_\mathcal{R}$ be the automaton for term $f(x, g(y))$. Let $\mathcal{A}$ be the automaton recognizing the language $\{f(a, g^*(b)), f(g^*(b), a)$. Let $\mathcal{A}_\cap$ be the intersection automaton between $\mathcal{A}_\mathcal{R}$ and $\mathcal{A}$. The automaton $\mathcal{A}_\cap$ recognizes, in final state (q_f, q_4), a unique $\mathcal{Q}$-instance $f((qx, q_1), g((qy, q_3))$ which defines the regular language substitution $\sigma = \{x \mapsto q_1,\ y \mapsto q_3\}$ and means that $f(x, g(y))\sigma \rightarrow_\mathcal{A}{}^* q_4$.*

$\mathcal{A}_\mathcal{R}$ with
$\mathcal{Q} = \{qx, qy, qf, qg\}$
$\mathcal{Q}_f = \{qf\}$
$\Delta =$

$$\begin{array}{rcl} y & \rightarrow & qy \\ g(qy) & \rightarrow & qg \\ x & \rightarrow & qx \\ f(qx, qg) & \rightarrow & qf \end{array}$$

$\mathcal{A}$ with
$\mathcal{Q} = \{q_1, q_2, q_3, q_4\}$
$\mathcal{Q}_f = \{q_4\}$
$\Delta =$

$$\begin{array}{rcl} a & \rightarrow & q_1 \\ b & \rightarrow & q_2 \\ g(q_2) & \rightarrow & q_3 \\ g(q_3) & \rightarrow & q_3 \\ f(q_1, q_3) & \rightarrow & q_4 \\ f(q_3, q_1) & \rightarrow & q_4 \end{array}$$

$\mathcal{A}_\cap$ with
$\mathcal{Q} = \{qx, qy, qf, qg\} \times \{q_1, q_2, q_3, q_4\}$
$\mathcal{Q}_f = \{qf\} \times \{q_1, q_2, q_3, q_4\}$
$\Delta =$ *(we omit useless transitions)*

$$\begin{array}{rcl} (y, q_2) & \rightarrow & (qy, q_2) \\ (y, q_3) & \rightarrow & (qy, q_3) \\ (x, q_1) & \rightarrow & (qx, q_1) \\ (x, q_3) & \rightarrow & (qx, q_3) \\ g((qy, q_2)) & \rightarrow & (qg, q_2) \\ g((qy, q_3)) & \rightarrow & (qg, q_3) \\ f((qx, q_1), (qg, q_3)) & \rightarrow & (qf, q_4) \\ f((qx, q_3), (qg, q_1)) & \rightarrow & (qf, q_4) \end{array}$$

Theorem 128 *Let $\mathcal{A} = \langle \mathcal{F}, \mathcal{Q}, \mathcal{Q}_f, \Delta \rangle$ be an automaton, $\mathcal{R}$ a TRS, $\mathcal{A}_\mathcal{R}$ the automaton of all left-hand sides of $\mathcal{R}$ and $\mathcal{A}_\cap$ the substitution automaton. Furthermore, we assume that $\mathcal{A}$ respects the left-coherence condition induced by $\mathcal{R}$. For all rewrite rule $l \rightarrow r \in \mathcal{R}$, for all state $q \in \mathcal{A}$, the set of regular language substitutions defined by the state $(top_\alpha(l), q)$ of $\mathcal{A}_\cap$ is exactly the set of regular language substitutions $\{\sigma_i\}_{i \in I}$ such that $l\sigma_i \rightarrow_\mathcal{A}{}^* q$.*

PROOF.

1. First assume that $\{\sigma_i\}_{i \in I}$ is a set of regular language substitutions defined by a term s such that $s \rightarrow^* (\alpha(l), q)$ in $\mathcal{A}_\cap$. For all σ_i, we prove that $l\sigma_i \rightarrow^* q$ by induction on the length of the derivations $s \rightarrow^n (\alpha(l), q)$:

   - If $s \rightarrow^1_{\mathcal{A}_\cap} (\alpha(l), q)$ then $depth(s) = 1$ and
     - either $s = (a, a)$ where a is a constant, we have $(a, a) \rightarrow_{\mathcal{A}_\cap} (\alpha(a), q)$ then $l = a$ and $l \rightarrow_\Delta q$, s defines the substitution of empty domain.
     - or $s = (x, a)$ we have also $(x, a) \rightarrow_{\mathcal{A}_\cap} (\alpha(a), q)$ then $l = x$, and s defines $\sigma_i = \{x \mapsto q\}$, we have $l\sigma = a$ and $a \rightarrow q$
   - Assume now that for all set $\{\sigma_i\}_{i \in I}$ defined by a term s of $\mathcal{A}_\cap$ verifying $s \rightarrow^k (\alpha(l), q)$ for every $k \leq n$ then $l\sigma_i \rightarrow^* q$ in $\mathcal{A}$.
     Consider now $\{\tau_i\}_{i \in I}$ defined by a term s such that $s \rightarrow^{n+1} (\alpha(l), q)$. Clearly s is a term of the form $f(s_1, \ldots, s_n)$, where $s_j = (l_j, q_j)$ define $\{\tau_i\}_{i \in I}$ such that $\tau_i = \sigma_{1,i_1} \circ \ldots \circ \sigma_{n,i_n}$ where σ_{j,i_j} range over the set of substitutions defined by the term s_j.
     $f(s_1, \ldots, s_n) \rightarrow_{\mathcal{A}_\cap} (\alpha(l), q)$ then l is of the form $f(l_1, \ldots, l_n)$ and there exist some states q_j verifying $s_j \rightarrow^k_{\mathcal{A}_\cap} (\alpha(l_j), q_j)$. We use the inductive hypothesis $s_j \rightarrow^k (\alpha(l_j, q_j)$ with $k \leq n$, then $l_j\sigma_j \rightarrow^* q_j$ in $\mathcal{A}$ and for all τ_i

there exists a combination of $\sigma_{i,j}$ verifying $l\tau_i \rightarrow f(l_1\sigma_{1,i_1}, \ldots, l_n\sigma_{n,i_n}) \rightarrow f(q_1, \ldots, q_n) \rightarrow^*_\Delta q$

2. We use now a structural induction over l to prove that if $l\sigma \rightarrow^*_\Delta q$ then σ correspond to a term s in $\mathcal{L}(\mathcal{A}_\cap)$:

   (a) if l is a constant a
   On the one hand $a \rightarrow_{\mathcal{A}_t} \alpha(a)$ and on the other hand there exists a state q in $\mathcal{Q}$ such that $a \rightarrow_{\mathcal{A}} q$ then $(a, a) \rightarrow_{\mathcal{A}_\cap} (\alpha(a), q)$ and $(\alpha(a), q)$ is a final state of $\mathcal{A}_\cap$, $\mathcal{A}_\cap$ recognized (a, a), that defines the empty substitution.

   (b) If l is a variable x
   $x \rightarrow_{\mathcal{A}_t} \alpha(x)$ and all the terms recognized by $\mathcal{A}$ may be an instance of x, we have $\{(x, q) \rightarrow (\alpha(x), q) | q \in \mathcal{Q}\} \subset \Delta_\cap$ and $\mathcal{Q}_{f,\cap} = \{\alpha(x)\} \times \mathcal{Q}$ we have defined all the regular language substitution $\{\{x \mapsto q\} | q \in \mathcal{Q}\}$.

   (c) If l is of the form $f(l_1, \ldots, l_n)$
   $\mathcal{A}$ recognized $l\sigma$ then there exist some states $q, q_1, \ldots q_n$ in $\mathcal{A}$ such that $f(q_1, \ldots, q_n) \rightarrow^*_{\mathcal{A}} q$, $l_i\sigma \rightarrow_{\mathcal{A}} q_i$. Assume that $s = f((\alpha(l_1), q_1), \ldots, (\alpha(l_n), q_n))$, there exist q_i in $\mathcal{A}$ and q_i recognized $t_i\sigma$ then thanks to induction hypothesis for each l_i there exists s_i, $\mathcal{Q}$-instance defining σ_i. If $\sigma_1 \circ \ldots \sigma_n$ is not defined then $l\sigma$ is not recognized by $\mathcal{A}$, contradiction ; else $\sigma_1 \circ \ldots \sigma_n$ exists and $\mathcal{A}_\cap$ recognized $s = f(s_1, \ldots, s_n)$ and s define σ.

□

4.2.3 Dealing with epsilon transitions and matching

In Timbuk 3.0, epsilon transitions are part of the completed automata. However the algorithm of Section 4.2.2 can easily be adapted so as to cover them. First, remark that epsilon transitions can only be encountered in the tree automaton to complete, *e.g.* $\mathcal{A}$, and not in $\mathcal{A}_\mathcal{R}$. The problems encountered when matching a term on a tree automaton with epsilon transitions are summed-up in the following example.

Example 129 *Let $\mathcal{A} = \langle \mathcal{F}, \mathcal{Q}, \mathcal{Q}_f, \Delta \rangle$, where $\mathcal{F} = \{f, g, a\}$, $\mathcal{Q} = \{q_0, q_1, q_2, q_3\}$, $\mathcal{Q}_f = \{q_0\}$ and $\Delta = \{f(q_1) \rightarrow q_0, g(q_2) \rightarrow q_2, q_2 \rightarrow q_1, q_3 \rightarrow q_2, \ldots\}$. Let $\mathcal{R} = \{f(g(x)) \rightarrow g(x)\}$. On the matching problem $f(g(x)) \trianglelefteq q_0$, a correct matching algorithm should give a substitution set $S = \{\{x \mapsto q_2\}, \{x \mapsto q_3\}\}$ since we have $f(g(q_2)) \rightarrow_{\mathcal{A}}{}^* q_0$ and $f(g(q_3)) \rightarrow_{\mathcal{A}}{}^* q_0$. However, if we use previous algorithm that do not take epsilon transitions into account it would answer $S = \emptyset$.*

To adapt the matching algorithm of Section 4.2.2 to tree automata with epsilon transitions, it is enough to extend the construction of the $\mathcal{A}_\cap$. It is similar to the one of Definition 123, except that $\Delta_\cap$ may contain also epsilon transitions.

Definition 130 (Substitution automaton (with epsilon transitions)) *If $\mathcal{A}$ contains epsilon transitions, $\mathcal{A}_\cap$ is defined as in Definition 123 except that definition of $\Delta_\cap$ is completed with a last case and becomes:*

$$\begin{aligned}\Delta_\cap = \quad & \{f((q_1,q'_1),\ldots,(q_n,q'_n)) \to (q_{n+1},q'_{n+1}) \mid q_i \in \mathcal{Q}_t,\ q'_i \in \mathcal{Q},\\ & \qquad f(q_1,\ldots,q_n) \to q_{n+1} \in \Delta_S,\ f(q'_1,\ldots,q'_n) \to q'_{n+1} \in \Delta\}\\ & \cup\{(x,q) \to (q_x,q) \mid x \to q_x \in \Delta_S,\ q \in \mathcal{Q}\}\\ & \cup(q,q_1) \to (q,q_2) \mid q \in \mathcal{Q}_S \ \wedge\ q_1,q_2 \in \mathcal{Q} \ \wedge\ q_1 \to q_2 \in \Delta\}\end{aligned}$$

◇

Other definitions can be kept as is. Like in the epsilon-free case, the algorithm obtained by computing the set of substitution induced by $\mathcal{Q}$-instance is exactly the set of valid regular language substitutions.

Example 131 *Let $\mathcal{A}_\mathcal{R}$ be the automaton for left-hand side of rule $f(g(x)) \to g(x)$. Let $\mathcal{A}$ be the tree automaton used in Example 129. Let $\mathcal{A}_\cap$ be the intersection automaton between $\mathcal{A}_\mathcal{R}$ and $\mathcal{A}$. The automaton $\mathcal{A}_\cap$ recognizes, in final state (qf,q_0), two $\mathcal{Q}$-instances $f(g(qx,q_2))$ and $f(g(qx,q_3))$ which define the regular language substitutions $\sigma = \{x \mapsto q_2\}$ and $\sigma' = \{x \mapsto q_3\}$ such that $f(g(x))\sigma \to_{\mathcal{A}}{}^* q_0$ and $f(g(x))\sigma' \to_{\mathcal{A}}{}^* q_0$.*

$\mathcal{A}_\mathcal{R}$ *with*	$\mathcal{A}$ *with*	$\mathcal{A}_\cap$ *with*
$\mathcal{Q} = \{qx, qf, qg\}$	$\mathcal{Q} = \{q_0,q_1,q_2,q_3\}$	$\mathcal{Q} = \{qx,qf,qg\} \times \{q_0,q_1,q_2,q_3\}$
$\mathcal{Q}_f = \{qf\}$	$\mathcal{Q}_f = \{q_0\}$	$\mathcal{Q}_f = \{qf\} \times \{q_0,q_1,q_2,q_3\}$
$\Delta =$	$\Delta =$	$\Delta =$ *(we omit useless transitions)*
$x \to qx$	$f(q_1) \to q_0$	$(x,q_2) \to (qx,q_2)$
$g(qx) \to qg$	$g(q_2) \to q_2$	$g((qx,q_2)) \to (qg,q_2)$
$f(qg) \to qf$	$q_2 \to q_1$	$f((qg,q_1)) \to (qf,q_0)$
	$q_3 \to q_2$	$(qg,q_2) \to (qg,q_1)$
	$\ldots$	$(qx,q_3) \to (qx,q_2)$

When automata include epsilon transitions, we can also remark that not all the computed substitutions are necessary for completion. If we look back at the previous example (Example 129), with the refined algorithm, we obtain the expected set of substitutions $S = \{\{x \mapsto q_2\},\{x \mapsto q_3\}\}$. Corresponding transitions to add are $T_1 = \{g(q_2) \to q', q' \to q_0\}$ and $T_2 = \{g(q_3) \to q'', q'' \to q_0\}$. However, one can remark that it is redundant to add T_2 once T_1 has been added since because of $q_3 \to q_2$ in $\mathcal{A}$, in the completed automaton we will have: $g(q_3) \to^\epsilon g(q_2) \to q' \to q_0$. This is a trivial critical pair (see Definition 100). Moreover, we can remark that the substitution $\{x \mapsto q_3\}$ has been added *because* of the transition $q_3 \to q_2$, we can deduce that there is no need to consider such substitutions during matching. During the construction of instances it is thus useless to consider rewritings with epsilon transitions of $\Delta_\cap$ on "variable states". In the previous example, this means that substitutions obtained with the instance $f(g(x,q_3)) \to f(g(qx,q_3)) \to f(g(qx,q_2)) \to \ldots$ are not necessary because of the use of transition $(qx,q_3) \to (qx,q_2)$ on state (qx,q_3). As a result, it is not necessary to add any transition of the form $(qx,q) \to (qx,q')$ when building $\mathcal{A}_\cap$.

This matching algorithm is implemented in Timbuk 3.0 using a specific oriented graph structure for epsilon transitions, i.e. a graph where states are vertices and epsilon transitions

are edges. We also use specific algorithm to compute the transitive closure of this graph incrementally after each adding of a new epsilon transition. This permits to efficiently resolve instance constructions with epsilon transitions of Δ_{ext} and also to avoid looping computations due to possible cycles in the epsilon graph of Δ_{ext}. Some other implementation details, as tabling of substitutions, are explained in the next section.

4.3 Implementation and use of matching in Timbuk

As explained above, the matching algorithm is the core of the completion implementation. This is particularly true in Timbuk 3.0 where matching is also used for detection of equations application (see Section 3.2.1) as well as for checking for forbidden patterns reachability (see Section 4.1.2). In previous section we have presented optimizations performed on the matching algorithm itself. Now, we present some hints used for efficient *implementation* of this algorithm.

4.3.1 Tabling and propagation of new substitutions

In Timbuk 3.0, substitutions are tabled. Every new found substitution for every state (not necessarily final) (q, q') of $\mathcal{A}_\cap$ is added to a table. To quickly construct new substitutions and only consider states of $\mathcal{A}_\cap$ where they may occur, the construction of $\mathcal{A}_\cap$ is performed incrementally.

Let $\mathcal{A}$ be a tree automaton, $\mathcal{R}$ be a TRS, $\mathcal{A}_\mathcal{R}^{i-1}$ (resp. $\mathcal{A}_\mathcal{R}^{i}$) be the $i-1$-th (resp. i-th) step completion automaton, $\mathcal{A}_\mathcal{R}$ be the tree automaton for left-hand sides of $\mathcal{R}$ and $\mathcal{A}_\cap^i$ be the intersection of automata $\mathcal{A}_\mathcal{R}$ and $\mathcal{A}_\mathcal{R}^i$. It is possible to incrementally construct $\mathcal{A}_\cap^i$ from $\mathcal{A}_\cap^{i-1}$ using $\mathcal{A}_\mathcal{R}$ and Δ_{Ext} the set of new transitions added to $\mathcal{A}^{i-1}$ to obtain $\mathcal{A}^i$. To obtain $\mathcal{A}_\cap^i$ from $\mathcal{A}_\cap^{i-1}$ it is enough to add to $\Delta_\mathcal{R} \times \Delta_{Ext}$ to $\Delta_\cap^{i-1}$. This product give a precise list of states where a new substitution can be found.

Thus, it is useless to compute from scratch the set of $\mathcal{Q}$-instance for every new $\mathcal{A}_\cap^i$ but, instead, propagate the information contained in $\Delta_\mathcal{R} \times \Delta_{Ext}$ bottom-up in order to determine new substitutions. We show how this is done on the following example.

Example 132 *Let $\mathcal{A}_\mathcal{R}$ be the automaton for TRS $\mathcal{R} = \{f(x) \to g(x), g(x) \to h(x)\}$ and let $\mathcal{A}^0 = \mathcal{A}$ be the tree automaton to complete. It recognizes the term $f(a)$. Thus, $\mathcal{A}_\cap^0$ is the intersection automaton between $\mathcal{A}_\mathcal{R}$ and $\mathcal{A}^0$.*

$\mathcal{A}_{\mathcal{R}}$ *with*
$\mathcal{Q} = \{qx, qf, qg\}$
$\mathcal{Q}_f = \{qf, qg\}$
$\Delta =$

$$\begin{array}{rcl} x & \rightarrow & qx \\ g(qx) & \rightarrow & qg \\ f(qx) & \rightarrow & qf \end{array}$$

$\mathcal{A}^0$ *with*
$\mathcal{Q} = \{q_0, q_1\}$
$\mathcal{Q}_f = \{q_0\}$
$\Delta =$

$$\begin{array}{rcl} f(q_1) & \rightarrow & q_0 \\ a & \rightarrow & q_1 \end{array}$$

$\mathcal{A}^0_\cap$ *with*
$\mathcal{Q} = \{qx, qf, qg\} \times \{q_0, q_1\}$
$\mathcal{Q}_f = \{qf, qg\} \times \{q_0, q_1\}$
$\Delta =$

$$\begin{array}{rcl} (x, q_1) & \rightarrow & (qx, q_1) \\ (x, q_0) & \rightarrow & (qx, q_0) \\ f((qx, q_1)) & \rightarrow & (qf, q_0) \end{array}$$

$\mathcal{A}^1$ *with*
$\mathcal{Q} = \{q_0, q_1, q_2\}$
$\mathcal{Q}_f = \{q_0\}$
$\Delta =$

$$\begin{array}{rcl} f(q_1) & \rightarrow & q_0 \\ a & \rightarrow & q_1 \\ & & \\ g(q_1) & \rightarrow & q_2 \\ q_2 & \rightarrow & q_0 \end{array}$$

$\mathcal{A}^1_\cap$ *with*
$\mathcal{Q} = \{qx, qf, qg\} \times \{q_0, q_1, q_2\}$
$\mathcal{Q}_f = \{qf, qg\} \times \{q_0, q_1, q_2\}$
$\Delta =$

$$\begin{array}{rcl} (x, q_1) & \rightarrow & (qx, q_1) \\ (x, q_0) & \rightarrow & (qx, q_0) \\ f((qx, q_1)) & \rightarrow & (qf, q_0) \\ & & \\ g((qx, q_1)) & \rightarrow & (qg, q_2) \\ (qf, q_2) & \rightarrow & (qf, q_0) \\ (x, q_2) & \rightarrow & (qx, q_2) \end{array}$$

From $\mathcal{A}^0_\cap$ we can find (and table) the association $(qf, q_0) \mapsto \{\{x \mapsto q_1\}\}$ meaning that there is a substitution for left-hand side of the rule $f(x) \rightarrow g(x)$ on state q_0, i.e. $f(q_1) \rightarrow^*_{\mathcal{A}^0} q_0$. To obtain $\mathcal{A}^1$ from $\mathcal{A}^0$ we add the transitions necessary to recognize $g(q_1)$ into q_0. Similarly to produce $\mathcal{A}^1_\cap$ from $\mathcal{A}^0_\cap$ we only add the 3 last transitions of $\mathcal{A}^1_\cap$. They come from product between $\mathcal{A}_{\mathcal{R}}$ and the new transitions of $\mathcal{A}^1$ (what we called Δ_{Ext} above). From this product, we directly obtain that there are only *new substitutions for the second rule (qg) on state q_2. In particular, the substitution found at previous step, i.e. first rule (qf) on q_0 is naturally not considered again.*

The above example illustrates how the construction is performed but not how propagation and combination of existing substitutions is optimized. An example of this could be the following. Assume that at step i, $\Delta_{\mathcal{R}} \times \Delta_{Ext} = \{f((q'_1, q_1)) \rightarrow (q', q)\}$. Since there is no new transition *leading* to (q'_1, q_1) at step i, we know that substitutions computed at step $j < i$ and tabled for (q'_1, q_1) are still valid, i.e. no new substitutions need to be computed for (q'_1, q_1). Let us call S_1 this set of substitution. Since the transition $f((q'_1, q_1)) \rightarrow (q, q')$ is in $\Delta_{\mathcal{R}} \times \Delta_{Ext}$ this means that above (q', q) there may be a new substitution. Now, assume that $\mathcal{A}^{i-1}_\cap$ contains the transition $g((q'_0, q_0), (q, q')) \rightarrow (q'_f, q_f)$. Like above, since $\Delta_{\mathcal{R}} \times \Delta_{Ext}$ does not contain any transition leading to (q'_0, q_0), we know that substitutions tabled for this state are still valid. Let us call S_2 this set of substitutions. What needs to be done, however, is to compute the combination of substitutions of S_1 and S_2 and to propagate them to (q'_f, q_f). Combination of sets S_1 and S_2 is simply the set $S = \{\sigma_1 \circ \sigma_2 \mid \sigma_1 \in S_1 \wedge \sigma_2 \in S_2\}$. If some substitutions of S are not already tabled for (q'_f, q_f) then they are added, propagated above, and so on. A new transition is produced, i.e. a substitution is applied, if it covers a complete left-hand side of a rule (and not only a subterm). This happens each time that a final state of $\mathcal{A}_\cap$ is met during new substitution propagation.

This technique permits to compute *only new* substitutions and trigger the addition of a new transition each time that a complete substitution is found.

4.3.2 Simplification of tree automata with equations

As said above, matching and tabling of substitution is performed for optimizing the detection of simplification positions, i.e. for any equation $s = t$ a regular language substitution σ and two distinct states q_1 and q_2 such that $s\sigma \rightarrow_{\mathcal{A}}{}^* q_1$ and $t\sigma \rightarrow_{\mathcal{A}}{}^* q_2$. Since detecting a rule application is very similar to detecting a simplification position, we use techniques similar to those presented in the previous section.

After having found a simplification position, recall that we applied merging on the tree automaton, i.e. replace $\mathcal{A}$ by $\mathcal{A}\{q_1 \mapsto q_2\}$ (see Section 3.2.1). The two obvious consequences of this renaming are a complete removal of state q_1 in $\mathcal{A}$ and a collapsing of transitions related to q_1 and q_2. An interesting property of the transition product is the following.

Proposition 133 *Let Δ_1 and Δ_2 be sets of transitions on distinct set of states and ρ a renaming of states of Δ_2.*

$$\Delta_1 \times (\Delta_2\rho) = (\Delta_1 \times \Delta_2)\rho$$

As a consequence, $\Delta_{\mathcal{R}} \times (\Delta\{q_1 \mapsto q_2\}) = (\Delta_{\mathcal{R}} \times \Delta)\{q_1 \mapsto q_2\}$. In other words, if we have $\mathcal{A}^i_\cap$ and a merging is applied to $\mathcal{A}^i$, then the substitution automaton between $\mathcal{A}_{\mathcal{R}}$ and the merged version of $\mathcal{A}^i$ can be obtained by applying the same merging to $\mathcal{A}^i_\cap$.

Another interesting property is that substitutions remain valid once renamed. The principle is to rename tables indexes and substitution sets. Assume that the table contains only three entries $(qf, q_1) \mapsto S_1$ and $(qf, q_2) \mapsto S_2$ and $(qg, q_1) \mapsto S_3$ and that the renaming is $\rho = \{q_1 \mapsto q_2\}$. The renamed table will contain: $(qf, q_2) \mapsto (S_1 \cup S_2)\rho$ and $(qg, q_2) \mapsto S_3\rho$. As in the previous section, this kind of renaming may produce new substitutions that have to be propagated above, etc.

4.4 Tree automata completion extensions

4.4.1 Discussion about non left-linear term rewriting systems

The algorithms proposed in Chapter 3 can straightforwardly deal with left-linear TRSs and with non left-linear TRS provided that tree automata and TRS fulfill the left coherence condition (see Definition 62). Let us recall what is the problem with tree automata completion on non left-linear rules and the two solutions proposed in Chapter 3. Let $f(x, x) \rightarrow g(x)$ be a rule of $\mathcal{R}$ and let $\mathcal{A}$ be a tree automaton whose set of transitions contains $f(q_1, q_1) \rightarrow q_0$ and $f(q_2, q_3) \rightarrow q_0$. Although we can find a valid substitution $\sigma = \{x \mapsto q_1\}$ and obtain a critical pair $f(q_1, q_1) \rightarrow q_0$ $g(q_1) \not\rightarrow q_0$ on the first transition, it is not the case for the second one since it is not possible to find a substitution mapping x to q_1 and to q_2. The semantics of a completion between rule $f(x, x) \rightarrow g(x)$ and transition $f(q_2, q_3) \rightarrow q_0$ would be to find the common language of terms recognized both by q_2 and q_3. There are two main solutions to this problem:

1. *Compute an intersection* between languages recognized by q_2 and q_3 and use the state recognizing this language in the substitution σ. This can be obtained by computing a new tree automaton $\mathcal{A}'$ with a set of states $\mathcal{Q}'$ such that $\mathcal{Q}'$ is disjoint from states of $\mathcal{A}$ and $\exists q \in \mathcal{Q}' : \mathcal{L}(\mathcal{A}', q) = \mathcal{L}(\mathcal{A}, q_2) \cap \mathcal{L}(\mathcal{A}, q_3)$. Then, to end the completion step it would be enough to add transitions of $\mathcal{A}'$ to $\mathcal{A}$ and the new transitions $g(q) \to q'$ and $q' \to q$.

2. *Determinize* tree automata. Indeed, one can remark that the non-linearity problem would disappear with deterministic automata since for any deterministic automaton $\mathcal{A}_{det}$ and for all states q, q' of $\mathcal{A}_{det}$ we trivially have $\mathcal{L}(\mathcal{A}, q) \cap \mathcal{L}(\mathcal{A}, q') = \emptyset$. However, determinization of a tree automaton may result in an exponential blow-up of the number of states (Comon et al., 2008). Furthermore note that the tree automata completion algorithm does not preserve deterministic tree automata. For instance, the tree automaton $\mathcal{A}$ with set of final states $\{q_0, q_1\}$ and set of transitions $\{a \to q_0, b \to q_1\}$ is deterministic, but if we apply completion w.r.t. TRS $\mathcal{R} = \{a \to b\}$ we obtain a new transition $q_1 \to q_0$. In the end, the completed tree automaton is no longer deterministic since it recognizes b into state q_0 *and* into state q_1.

In the literature, some papers consider non left-linear TRSs but the given solution is more theoretical than practical (Gyenizse and Vágvölgyi, 1998; Réty, 1999; Feuillade et al., 2004). For instance, in (Réty, 1999; Feuillade et al., 2004), the construction of $\mathcal{R}^*(\mathcal{L}(\mathcal{A}))$ step by step adds some transitions to $\mathcal{A}$, that have to be determinized (leading to exponential blow-up) if non left-linear rules are under concern. In (Genet and Viet Triem Tong, 2001) (mentioned also in Chapter 3), we proposed a solution between the two previous ones where, named *locally deterministic* tree automata where only a subset of states are kept deterministic. If we switch back to our previous example: rule $f(x, x) \to g(x)$ and transitions $f(q_1, q_1) \to q_0, f(q_2, q_3) \to q_0$, the aim is to only add the transitions $g(q_1) \to q, q \to q_0$ and keep q_2, q_3 deterministic. In the end of completion, in order to verify that the fixpoint automaton $\mathcal{A}^*_{\mathcal{R},E}$ is complete w.r.t. $f(x, x) \to g(x)$, it is necessary to check that $\mathcal{L}(\mathcal{A}^*_{\mathcal{R},E}, q_2) \cap \mathcal{L}(\mathcal{A}^*_{\mathcal{R},E}, q_3) = \emptyset$. In practice, this solution avoids the exponential blow-up of determinization while enabling the use of non left-linear TRSs. This was successfully used in the particular case of cryptographic protocols, see Section 5.1.

This solution, even if it is well adapted for specific needs, is only partial. To the best of our knowledge, the only paper giving a complete solution is the framework proposed by Takai et al. in (Takai et al., 2000). Takai et al. uses an algorithm, similar to completion, extended with so called *packed states* (see Definition 44) i.e. states representing intersections of languages. Non left-linear rules are applied by producing a tree automaton recognizing the intersection between the languages concerned by the non linear variable (case 1.). On our example with rule $f(x, x) \to g(x)$ and packed transitions $f(\langle q_1\rangle, \langle q_1\rangle) \to \langle q_0\rangle, f(\langle q_2\rangle, \langle q_3\rangle) \to \langle q_0\rangle$, Takai et al.'s algorithm produces new transitions $g(\langle q_1\rangle) \to \langle g(\langle q_1\rangle)\rangle, \langle g(\langle q_1\rangle)\rangle \to \langle q_0\rangle$ on one side and $g(\langle q_2, q_3\rangle) \to \langle g(\langle q_2, q_3\rangle)\rangle, \langle g(\langle q_2, q_3\rangle)\rangle \to \langle q_0\rangle$ on the other side. Each time that a new packed state, here $\langle q_2, q_3\rangle$ is found, their algorithm also adds the entire set of transitions recognizing terms in this packed state. In this case, it corresponds to adding the transitions of the tree automaton recognizing $\mathcal{L}(\mathcal{A}, q_2) \cap \mathcal{L}(\mathcal{A}, q_3)$. This corresponds to the first proposed solution. Although, it avoids the exponential blow-up of determinization, this algorithm may

produce very big tree automata in practice. Indeed, each application of a non left-linear rule may increase the size of the completed automaton by a quadratic factor.

A first optimization of packed states

While developing the version 3.0 of Timbuk, we experimented with a tree automata representation using packed states and non left-linear rules. We added several optimizations so as to avoid, whenever it is possible, to compute tree automata intersections. First, we remarked that completion produces a lot of trivial inclusion constraints that can be used. Indeed, the completion algorithm of Section 3.2.3 is based on critical pair solving of the form:

$$\begin{tikzcd} l\sigma \arrow[r, "\mathcal{R}"'] \arrow[d, "\mathcal{A}^i_{\mathcal{R}}"'] & r\sigma \arrow[d, "\mathcal{A}^{i+1}_{\mathcal{R}}"] \\ q & q' \arrow[l, "\mathcal{A}^{i+1}_{\mathcal{R}}"] \end{tikzcd}$$

From such a completion step, we can deduce that $\mathcal{L}(\mathcal{A}^{i+1}_{\mathcal{R}}, q') \subseteq \mathcal{L}(\mathcal{A}^{i+1}_{\mathcal{R}}, q)$, hence that $\mathcal{L}(\mathcal{A}^{i+1}_{\mathcal{R}}, q') \cap \mathcal{L}(\mathcal{A}^{i+1}_{\mathcal{R}}, q) = \mathcal{L}(\mathcal{A}^{i+1}_{\mathcal{R}}, q')$. In other words, for achieving a completion step with a rule $f(x, x) \to g(x)$ on a transition $f(q_2, q_3) \to q_0$, if we know that $q_3 \to^*_\epsilon q_2$, then it is not necessary to build a tree automaton recognizing the intersection between q_2 and q_3, since it is already present in the tree automaton and recognized by q_3. Using packed states, this would mean that $\langle q_2, q_3\rangle = \langle q_3\rangle$.

However, this simple optimization is not enough to solve the intersection problems in the general case and it is often necessary to add transitions recognizing the intersection.

Example 134 *Assume that we have the following transitions in a completed tree automaton:*

$$\begin{tikzcd} & f \arrow[r] \arrow[dl, dash] \arrow[dr, dash] & q & & f \arrow[r] \arrow[dl, dash] \arrow[dr, dash] & q' \\ q_1 \arrow[rrr, bend right, "\epsilon"'] & & q_2 & q_3 \arrow[lll, bend left, "\epsilon"] & & q_4 \end{tikzcd}$$

Then, we clearly have $\langle q_1, q_3\rangle = \langle q_1\rangle$ and $\langle q_2, q_4\rangle = \langle q_4\rangle$. In order to recognize the intersection language of state $\langle q, q'\rangle$ it is necessary to add the transition $f(\langle q_1\rangle, \langle q_4\rangle) \to \langle q, q'\rangle$.

A second optimization of packed states

When we create a new packed state recognizing an intersection language, i.e. a packed state with at least two states, it is not always necessary to add the corresponding transitions to the tree automaton. This can be done in a lazy way, by producing the transitions when a rewrite rule has to be matched into a packed state. More precisely, for any rewrite rule $l \to r$ and any packed state, say $\langle q_1, q_2\rangle$, of a tree automaton $\mathcal{A}$:

- it is *never* necessary to produce transitions for $\langle q_1, q_2 \rangle$ to match l over $\langle q_1, q_2 \rangle$ (on top position). The reason is simple. Assume that there exists a substitution σ such that $l\sigma \rightarrow_{\mathcal{A}}{}^* \langle q_1, q_2 \rangle$. Since the packed state $\langle q_1, q_2 \rangle$ recognizes the intersection of languages recognizes by q_1 and q_2, there exist substitutions σ_1 and σ_2 such that $l\sigma_1 \rightarrow_{\mathcal{A}}{}^* q_1$, $l\sigma_2 \rightarrow_{\mathcal{A}}{}^* q_2$ and the intersection of languages corresponding to $l\sigma_1$ and $l\sigma_2$ is $l\sigma$. In that case, completion adds transitions $r\sigma_1 \rightarrow q'_1, q'_1 \rightarrow q_1$ and $r\sigma_2 \rightarrow q'_2, q'_2 \rightarrow q_2$. It is easy to see that, in that case, the intersection of the languages corresponding to $r\sigma_1$ and $r\sigma_2$ is recognized, implicitly by packed state $\langle q'_1, q'_2 \rangle$. In other words, the completion of the intersection of states is obtained by the intersection of each state on which completion has been applied separately. As a result, doing completion on intersection packed states would introduce redundant information in the tree automaton.

- if l is of the form $f(\ldots g(\ldots)\ldots)$ and we have a transition $f(\ldots \langle q_1, q_2 \rangle \ldots) \rightarrow q_0$ then it is necessary to produce all the transitions recognizing $\langle q_1, q_2 \rangle$ before trying to match $f(\ldots g(\ldots)\ldots)$ on $f(\ldots \langle q_1, q_2 \rangle \ldots) \rightarrow q_0$.

Conclusion on the experiments with non left-linearity

The experiments we made during the development of Timbuk 3.0 showed that dealing with non left-linear rules add a great overhead to the completion times. Even more annoying, we realized that optimizing the treatment of inclusion constraints was bringing down the overall performances of the completion even for left-linear rules.

On a more pragmatic point of view, we studied the general form of non left-linear rules we needed in the modelization of cryptographic protocols or Java bytecode programs (see Section 5.1 and Section 5.2). We made two observations. The first one is that in those two formal models we needed very few non left-linear rules. In the case of Java bytecode programs, it appeared that, with rather small effort, it was even possible to avoid them totally. The second one is that non left-linear rules we needed were generally of the form: $f(x, x, y) \rightarrow g(y)$. This is the case, for instance, for the general knowledge deduction rule used by an intruder in cryptographic protocols verification: $store(encr(k, m), k) \rightarrow m$. This rule means that if an intruder knows a message m ciphered by a key k and he also knows k then he is able to deduce m in clear. Similarly, one can find a rule for the definition of the $\otimes$ operator: $x \otimes x \rightarrow 0$. Those two rules share a common form $f(x, x, y) \rightarrow g(y)$ that checks that the same x is found at position 1 and 2 but then only use the y value. Applying completion on such a rewrite rule and the transition $f(q_1, q_2, q_3) \rightarrow q_0$ thus only require to check that the intersection of languages recognized by q_2 and q_3 is not empty and then to add the transitions $g(q_3) \rightarrow q', q' \rightarrow q_0$. It does not require to add to the completed automaton any transitions for language intersection between q_1 and q_2.

Recall that adding language intersections may increase the size of the tree automaton by a quadratic factor at each step of completion. This is the reason why, in Timbuk 3.0, we finally chose only to cover the particular case of non left-linearity shown above, and thus preserve the overall performances of the completion. In order to make this restriction explicit in the syntax of Timbuk specifications, we chose to forbid non left-linear rules and accept a restricted form of conditional rules encoding the above non left-linearity form.

4.4.2 Restricted conditional term rewriting systems

Timbuk 3.0 accepts conditional rules of the form $l \to r$ if $x_1 \downarrow x'_1$ and $\ldots$ and $x_n \downarrow x'_n$ where $x_1, x'_1, \ldots, x_n, x'_n \in \mathcal{V}ar(l)$. Completion using these conditional rules is defined according to the theoretical framework of Section 3.1.4. Any non-left linear rule of the form $f(x, x, y) \to g(y)$ can thus be encoded using a conditional rule $f(x, x', y) \to g(y)$ if $x \downarrow x'$. Note also that any rule of the form $f(x, x) \to g(x)$ will necessarily be encoded by a conditional rule of the form $f(x, x') \to g(x)$ if $x \downarrow x'$ (or $f(x, x') \to g(x')$ if $x \downarrow x'$) making explicit the fact that no intersection is computed by completion. Indeed, assume that we have a transition $f(q_2, q_3) \to q_0$ in the tree automaton, the critical pair would thus add $g(q_2) \to q$ (or $g(q_3) \to q$) and $q \to q_0$. Of course, this is less precise than computing the intersection of languages recognized by q_2 and q_3 but, how it is explained in the previous section, it keeps completed automata size reasonable in practice.

Limitation of conditional term rewriting systems w.r.t. non left-linear rules

In Section 3.1.4, we have given the theory about completion with general conditional TRSs. Let $f(x, x') \to g(x)$ if $x \downarrow x'$ be the conditional rule to apply and $f(q_2, q_3) \to q_0$ be the transition of the tree automaton to be considered. By definition of completion with conditional rules, the transitions $g(q_2) \to q'$ and $q' \to q_0$ will be added if and only if $\mathcal{L}(\mathcal{A}, q_2) \cap \mathcal{L}(\mathcal{A}, q_3) \neq \emptyset$. This reveals another weakness of the encoding of non left-linear rules into conditional rules. Indeed, this encoding is not convenient for non left-linear rules where there are strictly more than 2 occurrence of the same variable. For instance, a rule of the form $f(x, x, x, y) \to g(x)$, though it can exactly be encoded using a conditional rule $f(x, x', x'', y) \to g(x)$ if $x \downarrow x'$ and $x' \downarrow x''$ when rewriting terms, this is no longer the case with completion. For instance, a reasonable semantics for completion of a transition $f(q_2, q_3, q_4, q_5) \to q_0$ w.r.t. rule $f(x, x, x, y) \to g(y)$ would be to add transitions $g(q_5) \to q'$ and $q' \to q_0$ if and only if

$$(1)\mathcal{L}(\mathcal{A}, q_2) \cap \mathcal{L}(\mathcal{A}, q_3) \cap \mathcal{L}(\mathcal{A}, q_4) \neq \emptyset.$$

When using the above encoding in conditional rewrite rule $f(x, x', x'', y) \to g(x)$ if $x \downarrow x'$ and $x' \downarrow x''$, the checking becomes:

$$(2)\mathcal{L}(\mathcal{A}, q_2) \cap \mathcal{L}(\mathcal{A}, q_3) \neq \emptyset \text{ and } \mathcal{L}(\mathcal{A}, q_3) \cap \mathcal{L}(\mathcal{A}, q_4) \neq \emptyset).$$

Of course, (1) implies (2) (and thus we still have an over-approximation) but they are not equivalent. However, on the practical cases we had, no such non left-linear rules with 3 or more occurrences are necessary.

Now, we give some details about the implementation of completion with these restricted conditional rules. The optimization of matching of linear left-hand sides of rules on the tree automaton is detailed in Section 4.2. Thus, we here focus on the optimization of the test of non-emptiness of the intersection, i.e. $\mathcal{L}(\mathcal{A}, q) \cap \mathcal{L}(\mathcal{A}, q') \neq \emptyset$ for given q and q'.

For this test, we neither build the intersection automaton nor check for emptiness but achieve both at once. Let Δ and $\mathcal{Q}$ be respectively the set of transitions and the set of states of $\mathcal{A}$. The algorithm uses two sets $ok \subset \mathcal{Q} \times \mathcal{Q}$ and $rec \subset \mathcal{Q} \times \mathcal{Q}$ and a recursive function $check : \mathcal{Q} \times \mathcal{Q} \mapsto bool$. The call $check(q, q')$ answers $true$ if $\mathcal{L}(\mathcal{A}, q) \cap \mathcal{L}(\mathcal{A}, q') \neq \emptyset$.

The objective of the algorithm is to find, as quickly as possible, a common term between states q and q' or on the opposite to prove rapidly that no such term exists.

The general principle of $check(q, q')$ is the following: if there are two transitions $f(q_1, \ldots, q_n) \to q$ and $f(q'_1, \ldots, q'_n) \to q'$ and $check(q_1, q'_n) = true, \ldots, check(q_n, q'_n) = true$ then $check(q, q') = true$. Used as is, this function may not terminate on recursive transition sets. The reason for non termination is the following. Assume that we want to check non emptiness of the intersection between q and q' and that Δ contains transitions $f(q) \to q$ and $f(q') \to q'$ then this recursive algorithm may go on forever. Note however, that those recursive transitions are not necessary to consider for non-emptiness decision. Indeed, in the previous case (q, q') is not empty *only if* some other transitions contribute to the languages recognized by states q and q'. This is the reason why we use the rec state for tabling couples that have already been recursively inspected.

This general principle needs to be completed with several optimization for better efficiency. First, we use the ok set to table all couple of states whose intersection has already proven non empty. Second, there are some cases where $check(q, q')$, and thus $\mathcal{L}(\mathcal{A}, q) \cap \mathcal{L}(\mathcal{A}, q') \neq \emptyset$, can be computed without recursive calls by a careful inspection of the set of epsilon transitions or a careful inspection of Δ. Here is a more detailed description of the recursive $check$ function.

$check(q, q') =$

1. if $(q, q') \in ok$ then return $true$
2. if $(q, q') \in rec$ then return $false$
3. if $q \to_{\mathcal{A}}{}^* q'$ or $q' \to_{\mathcal{A}}{}^* q$ then $ok := (q, q') \cup ok$; return $true$[1]
4. if there exists at least a common constant symbol $a \in \mathcal{F}$ such that $a \to q \in \Delta$ and $a \to q' \in \Delta$ then $ok := (q, q') \cup ok$; return $true$
5. $rec := rec \cup (q, q')$
6. for all functional symbol $f \in \mathcal{F}$ of arity n such that $f(q_1, \ldots, q_n) \to q \in \Delta$ and $f(q'_1, \ldots, q'_n) \to q' \in \Delta$ do
   - if $check(q_1, q'_1)$ and $\ldots$ and $check(q_n, q'_n)$ then $ok := (q, q') \cup ok$; return $true$

   done
7. otherwise return $false$

After each completion step, values of ok and rec are initialized to $\emptyset$ and the $check$ function is evaluated for all conditions to check. Note that the value of the ok set could be used from one step to another but this is not the case in the current implementation. Indeed, on the one side, completion only adds transitions to states. On the other side, state couples of ok could be renamed according to state merging operation due to simplification with equations. For

[1] provided that corresponding states are filled (see Definition 94) which we always assume on all states of all tree automata we consider. Note that, this property is guaranteed on $\mathcal{R}/E$-coherent tree automata.

instance if $(q, q') \in ok$, i.e. $\mathcal{L}(\mathcal{A}, q) \cap \mathcal{L}(\mathcal{A}, q') \neq \emptyset$, and q' is merged with q'' we trivially have $\mathcal{L}(\mathcal{A}, q) \cap \mathcal{L}(\mathcal{A}, q'') \neq \emptyset$ and we can simply replace (q, q') by (q, q'') in the ok set so that it remains valid.

4.5 Tree automata completion checker

Like many verification tools, Timbuk is a complex programs and a single error may jeopardize the entire trust chain. Efforts have been made to certify static analyzers (Klein and Nipkow, 2003; Barthe and Dufay, 2004; Cachera et al., 2005) or to certify the results obtained by static analyzers (Letouzey and Théry, 2000; Besson et al., 2006) in Coq(Bertot and Castéran, 2004) in order to increase confidence in the analyzers. In (Boyer et al., 2008), we have instantiated the general framework of (Besson et al., 2006) to the particular case of analyzing term rewriting systems by tree automata completion. The proof that the tree automaton produced by completion recognizes an over-approximation of all reachable terms is automatically done within the Coq proof assistant. Coq is based on constructive logic (Calculus of Inductive Constructions) and it is possible to extract an Ocaml function implementing exactly the algorithm whose specification has been expressed in Coq. The extracted code is thus a *certified* implementation of the specification given in the Coq formalism. Extracted programs are standalone and do not require the Coq environment to be executed. For details about the extraction mechanisms, readers can refer to (Bertot and Castéran, 2004).

A specific challenge is to marry constructive logic and efficiency. The efficiency of a tree automata completion defined in constructive logic would not be sufficient for real size problems. Case studies with tree automata completion, on cryptographic protocols (see Section 5.2) and on Java bytecode (see Section 5.1) show that we need an efficient completion algorithm to verify properties on this kind of models. As a result, the current implementation of completion, Timbuk 3.0 is based on imperative data structures like hash tables whereas Coq allows only pure functional structures. A second problem is the termination of completion. Since Coq can only deal with total functions, functions must be proved terminating for any computation. In general, such a property cannot be guaranteed on completion because it mainly depends on term rewriting system and approximation equations given initially.

For these two reasons, there is little hope to specify and certify an efficient and purely functional version of the completion algorithm. Instead, we have adopted a solution based on a result-checking approach. It consists of building a smaller program (called the *checker*) - certified in Coq - that checks if the tree automaton computed by Timbuk is sound. In (Boyer et al., 2008), we studied the case of left-linear term rewriting systems which revealed to be sufficient for verifying Java programs (Boichut et al., 2007).

The extracted checker, proposed in (Boyer et al., 2008), is able to *decide* if the given tree automaton is a valid fixpoint of tree automata completion. A great advantage of using an external checker is that it is totally independent of the completion tool we use. In particular, the implementation of the completion tool can be optimized as necessary: as long as it outputs a tree automaton, this result can be certified by our checker. Another interesting consequence is that the checker is able to certify fixpoints computed by other completion tools like those presented in Section 4.6.2.

From the Coq formal specification (about 2000 lines for definitions and 5500 lines for proofs), we have extracted an Ocaml checker implementation which is connected to the Timbuk parser. Since Coq extraction ignore all Ocaml data types (integers, lists, maps...) and redefine all them (including primitive types), we defined a set of functions to convert Coq types into Ocaml types and conversely.

In the following table, we have collected several benchmarks showing that certification is possible and, in fact, very efficient. For each test, we give the size of the two tree automata (initial $\mathcal{A}^0$ and completed $\mathcal{A}^*_{\mathcal{R}}$) as number of transitions/number of states. For each TRS $\mathcal{R}$ we give the number of rules. The 'CS' column gives the number of completion steps necessary to complete $\mathcal{A}^0$ into $\mathcal{A}^*_{\mathcal{R}}$ and 'CT' gives the completion time. The 'CKT' column gives the time for the checker to certify the $\mathcal{A}^*_{\mathcal{R}}$ and the 'CKM' gives the memory usage. The important thing to observe here is that, the completion time is very long (sometimes more than 24 hours), the *checking* of the corresponding automaton is always fast (a matter of seconds).

The four tests are Java programs translated into term rewriting systems using the technique detailed in Section 5.2. All of them are completed using Timbuk 2.2 except the example `List2.java` which has been completed using another optimized completion tool: a Tom-based completion tool detailed in Section 4.6.2. Even if the input and output of this tool are tree automata, the internal computation mechanism is exclusively based on term rewriting and uses no tree automata algorithms. This shows that the completed automaton produced by a totally different algorithm and fully optimized tool is also accepted by the checker. The `List1.java` and `List2.java` corresponds to the same Java program but with slightly different encoding into TRS and approximations. The `Ex_poly.java` is the example given in (Boichut et al., 2007) and the `Bad_Fixp` is the same problem as `Ex_poly.java` except that the completed automaton $\mathcal{A}^*_{\mathcal{R}}$ has been intentionally corrupted. Thus, this is thus not a valid fixpoint and rejected by the checker.

Name	$\mathcal{A}^0$	$\mathcal{A}^*_{\mathcal{R}}$	$\mathcal{R}$	CS	CT	CKT	CKM
`List1.java`	118/82	422/219	228	180	$\approx$ 3 days	0,9s	2,3 Mo
`List2.java`	1/1	954/364	308	473	1h30	2,2s	3,1 Mo
`Ex_poly.java`	88/45	951/352	264	161	$\approx$ 1 day	2,5s	3,3 Mo
`Bad_Fixp`	88/45	949/352	264	161	$\approx$ 1 day	1,6s	3,2 Mo

4.6 Comparison with other tools

We here compare Timbuk with other available tools. We first compare with Autowrite (Durand, 2006) which implements some techniques described in Chapter 2. Then, we briefly present two other completion tools, one based on Bddbddb (Whaley et al., 2004) and one on Tom (Tom, 2009). Finally, we compare with Maude (Clavel et al., 2009) with and without equational abstraction.

As far as we know the only other tool dealing with a TRS class of Section 2.1.1 is Autowrite developed by Irène Durand (Durand, 2005, 2006) for the **LL-G**$^{-1}$ class. We will compare Timbuk and Autowrite for the computation of reachable terms on this class.

For tree transducers and static analysis techniques based on regular languages, no tool comparison will be done. For tree transducers, as explained in Section 3.3.4, comparison

of tree transducer models with TRS models is, anyway, difficult. Additionally, as far as we know there is no distributed implementation of the abstract regular tree model-checking as defined in Section 3.3.4. There exists some other tools, like SPADE (Patin et al., 2007b) or PRESS (Sighireanu and Touili, 2006), dedicated to the reachability analysis of systems and using tree automata to represent reachable states. But, those tools run on program models which are different from tree transducers and are, themselves built over the Timbuk library to implement tree automata operations. Finally, with regards to static analysis techniques using regular languages, it seems that no implementation of techniques described in Section 2.3.2 is available.

4.6.1 On regular classes

As explained above, Autowrite is one of the rare available tools able to deal with some of the classes of Section 2.1.1. Autowrite (Durand, 2005) is in fact a very complete tool to manipulate Tree Automata and TRS. In particular, it implements all the operation that may be desirable on tree automata: intersection, union, emptiness decision, determinization, inclusion, equality, etc. The implementation of all those basic operations is generally much more efficient in Autowrite than in the Timbuk 2.2 library.

For a given left-linear growing TRS $\mathcal{R}$, Autowrite permits to compute set of ancestors of a given regular set of terms. Since Autowrite is essentially used for *call by need* analysis, it is generally used to compute ancestors of the set $IRR(\mathcal{R})$ which is regular if $\mathcal{R}$ is left-linear. This can be seen as computing the set $(\mathcal{R}^{-1})^*(IRR(\mathcal{R}))$. This is, in fact, the set of terms *having a normal form*, i.e. weakly normalizable terms. For this purpose, Autowrite approximates any TRS by a TRS for which this set can be computed exactly. This approximation, directly performed on the TRS, over-approximates its set of weakly normalizable terms. Several kind of approximations are possible among which the *growing* approximation (Jacquemard, 1996). The growing approximation consists in replacing variables of the left-hand side by new variables if they occur at depth 2 or more in the right-hand side. One can see that the growing approximation transforms any TRS in a TRS of the class **RL-G** of Section 2.1.1. The growing approximation is the most precise approximation proposed by Autowrite.

Using the approximation is generally not well suited for reachability analysis because it is very rough. However, we can compare the efficiency of Autowrite and Timbuk to compute $(\mathcal{R}^{-1})^*(IRR(\mathcal{R}))$. Autowrite is also able to deal with left-linear growing TRS, possibly being non right-linear. In that case, $\mathcal{R}^{-1}$ can be non left-linear. This raises the same problem as explained in Section 4.4.1. However, in the particular case of growing TRSs, it is possible to keep the tree automaton recognizing $(\mathcal{R}^{-1})^*(IRR(\mathcal{R}))$ deterministic. The construction is more complex but is efficiently implemented in Autowrite. Since Timbuk does not deal with deterministic tree automata we are going to consider only linear growing TRSs.

On nearly every example provided in its distribution, Autowrite constructs $(\mathcal{R}^{-1})^*(IRR(\mathcal{R}))$ within milliseconds. In the table of Figure 4.4, we gather the computation times for some more meaningful examples having a computation time greater to a second. The examples $\mathcal{R}_1$, $\mathcal{R}_4$ and $\mathcal{R}_5$ come from (Durand, 2005), Fib is a TRS computing Fibonnacci numbers and HL is computing prime numbers. All those examples are distributed with Autowrite.

	Autowrite			Timbuk 3.0		
TRS	Time (secs)	# states	# transitions	Time (secs)	# states	# transitions
$\mathcal{R}_5$	1.12	5	667	0.52	16	108
$\mathcal{R}_1$	3.56	26	1357	1.45	173	836
$\mathcal{R}_4$	8.55	11	1608	8.97	1627	3344
Fib	146.33	12	734	1.21	77	742
HL	-	-	-	4475.06	86	6339

Figure 4.4: Computation of $(\mathcal{R}^{-1})^*(IRR(\mathcal{R}))$ with Autowrite and Timbuk 3.0

Note that using TRSs in the reverse order and with growing approximations has some strong consequences on $\mathcal{R}^{-1}$. In particular all rules of the form $g(x) \to x$ in $\mathcal{R}$ become $x \to g(x)$ in $\mathcal{R}^{-1}$. These are no longer regular rewrite rules and they are not handled by Timbuk. Those rules are unfolded on any symbol $f \in \mathcal{F}$. In other words, all rules like above are replaced by a finite set of rules $\{f(x_1, \ldots, x_n) \to g(f(x_1, \ldots, x_n)) \mid f \in \mathcal{F} \text{ and } \mathcal{A}r(f) = n\}$. Moreover, with growing TRS, sets of variables do not necessarily coincide between the left and right-hand sides. For instance, we can have rules of the form $f(g(x)) \to h(g(y))$. In this case, y can have any possible value in $\mathcal{T}(\mathcal{F})$. In Timbuk we can take this behavior into account using a trick that is common to all the theoretical works and tools. This trick has been also used in Section 3.2.4 for Corollary 112. It consists in using a specific state, say $q_{\mathcal{T}(\mathcal{F})}$, and to add transitions $\{f(q_{\mathcal{T}(\mathcal{F})}, \ldots, q_{\mathcal{T}(\mathcal{F})}) \to q_{\mathcal{T}(\mathcal{F})} \mid f \in \mathcal{F}\}$ to the automaton to complete. Then, during completion, every occurrence of a variable y that cannot be instantiated by a state is replaced by $q_{\mathcal{T}(\mathcal{F})}$.

For each example, we computed the $IRR(\mathcal{R})$ automata using Autowrite and completed it using Autowrite, on one side and Timbuk 3.0 on the other side. Those experiments were carried out on a 2.16 Ghz Intel Core Duo computer. The results are gathered in the table of Figure 4.4. Additionally to the computation times, the table also gives the number of states and transitions of the automaton recognizing the growing approximation of $(\mathcal{R}^{-1})^*(IRR(\mathcal{R}))$. For each example, when the example files contain terms which are supposed to belong to $(\mathcal{R}^{-1})^*(IRR(\mathcal{R}))$ or not, it is checked on both results. For instance, the fib constant term in the Fib TRS rewrites to the infinite list of Fibonacci numbers. It cannot be normalized in any way. We can check that it does not belong to the linear growing approximation of $(\mathcal{R}^{-1})^*(IRR(\mathcal{R}))$. On the opposite, the term $nth(s(s(s(s(s(0))))), fib)$ which computes the 5-th Fibonacci number from the infinite list *can* be finitely rewritten. It is thus normalizable. Again, we were able to check that it belongs to the approximation of $(\mathcal{R}^{-1})^*(IRR(\mathcal{R}))$ with both tools.

The number of states and transitions obtained by Timbuk is generally greater that what is obtained by Autowrite because it does not use an algorithm dedicated to growing TRSs but the exact completion strategy, see Section 3.3.2, which covers more that just growing TRSs. In particular, in Timbuk 3.0 it adds a renamed copy of the initial tree automaton to the completed one. This partially explains the difference between, for instance, the 11 states and 1608 transitions for Autowrite and 1627 states and 3344 transitions for Timbuk 3.0 on the $\mathcal{R}_4$ example. In fact, after removing all the states that are not reachable and thus useless, the Timbuk tree automaton contains only 31 states and 1654 transitions.

For the last benchmark HL, there is no value for Autowrite because computation was manually stopped after 10 hours. On some other Autowrite examples, we had to stop both Autowrite and Timbuk. The fact that rather small TRSs exhaust those tools is not really surprising. This is due to the fact that each computation starts from the tree automaton $IRR(\mathcal{R})$ which is very combinatorial and whose size can be exponential w.r.t. the size of $\mathcal{R}$.

4.6.2 Other tree automata completion tools

We briefly present two other completion tools: one encoding the completion into horn clauses and an other one into pure rewriting. Surprisingly, though the two encodings were independently defined, they are very similar. They rely on the slicing of both left and right-hand sides of rewrite rules.

Tree automata completion by horn clauses

The first technique, proposed by (Gallagher and Rosendahl, 2008), is based on an encoding of both tree automata and term rewriting systems into Horn clauses. The rewriting symbol $\rightarrow$ used for rewriting and tree automata is encoded by a binary first order predicate _ $\rightarrow$ _. A tree automaton transition $f(q_1, \ldots, q_n) \rightarrow q$ is represented by the horn clause $(f(q_1, \ldots, q_n) \rightarrow q) \leftarrow true$. A rewrite rule $plus(s(x), y) \rightarrow s(plus(x, y))$ will be replaced by the following horn clauses:

$$(1)\ (plus(X, Y) \rightarrow q(X, Y)) \leftarrow (s(X) \rightarrow Q_1), (plus(Q_1, Y) \rightarrow Q)$$
$$(2)\ (s(q(X, Y)) \rightarrow Q) \leftarrow (s(X) \rightarrow Q_1), (plus(Q_1, Y) \rightarrow Q)$$

Now, let us explain how application of horn clauses on an initial tree automaton models completion. Assume that the tree automaton contains the following transitions:

$plus(q_1, q_2) \rightarrow q_0$	encoded by the facts	$(3)\ (plus(q_1, q_2) \rightarrow q_0) \leftarrow true$
$s(q_3) \rightarrow q_1$	(unit horn clauses)	$(4)\ (s(q_3) \rightarrow q_1) \leftarrow true$

Using the clauses (1) and (2) representing the TRS and the facts (3) and (4), we can thus deduce the following facts. Using clause (1) (resp. (2)) with $X = q_3$, $Q_1 = q_1$ $Y = q_2$ and $Q = q_0$, we obtain the fact (5) (resp. (6)):

$$(5)\ (plus(q_3, q_2) \rightarrow q(q_3, q_2)) \leftarrow true$$
$$(6)\ (s(q(q_3, q_2)) \rightarrow q_0) \leftarrow true$$

which is exactly the set of transitions that would have been obtained using completion and normalization with a new state named $q(q_3, q_2)$. However, like for completion the construction of the least Herbrand model is unlikely to terminate. The approach developed in (Gallagher and Rosendahl, 2008) computes an approximation using state-of-the-art static analysis tools for logic programs, i.e. the Bddbddb tool (Whaley et al., 2004). This results into a very efficient completion tool as shown in the table Figure 4.5. However, as for other static analysis tools (see Section 2.3.2) the approximation precision is fixed. This is different from what can be done using normalization rules or equations.

Tree automata completion by rewriting

The second technique, proposed in (Balland et al., 2008), encodes completion into the Tom rewriting tool (Tom, 2009). The encoding of tree automata uses a special variadic symbol $sons$ for representing sets of terms. For instance, the set of tree automata transitions $\{f(q_1, q_2) \to q, g(q) \to q\}$ is represented using a constant q that is associated to the term $sons(f(q_1, q_2), g(q))$. Then, as in the previous work, left-hand sides of rewrite rules are sliced as follows. Given $f(x, g(y)) \to f(g(x), g(y))$ a rewrite rule, its sliced encoding is

$$\begin{array}{l} g(y) \to C_1(y) \\ f(x, C_1(y)) \to C_2(x, y) \end{array}$$

Taking advantage of Tom's matching algorithm on variadic symbols, those rules can be replaced by the following set of Tom's rules:

$$\begin{array}{l} (1)\, g(y) \to C_1(y) \\ (2)\, f(x, sons(_*, C_1(y), _*)) \to C_2(x, y) \\ (3)\, C_2(x, y) \to f(g(x), g(y)) \end{array}$$

Now, we briefly explain how completion works. Let $q = sons(f(q_1, q_2))$ and $q_2 = sons(g(q_3))$ be the initial tree automaton. Rule (1) can be applied on the term q_2 that becomes: $q_2 = sons(g(q_3), C_1(q_3))$. Hence, rule (2) can be applied on term q because q can be unfolded as $sons(f(q_1, sons(g(q_3), C_1(q_3)))$ and the term $f(x, sons(_*, C_1(y), _*))$ matches the subterm at position $1.\epsilon$ such that $x = q_1$ and $y = q_3$. Hence, q becomes $sons(f(q_1, q_2), C_2(q_1, q_3)$. The next rewriting step consists in rewriting $C_2(q_1, q_3)$ into $f(g(q_1), g(q_3))$. Like in usual tree automata completion, this term needs to be normalized first by introducing new terms (in place of the new states of completion), i.e. terms $q_4 = sons(g(q_1))$. Hence q becomes $sons(f(q_1, q_2), C_2(q_1, q_3), f(q_4, q_2)$.

Comparison of the efficiency of the different completion techniques

The two encodings presented above lead to very efficient implementation of completion. We here give a comparison of the overall performances of the different tools implementing completion on some typical examples. In the table Figure 4.5, the automaton size is given as (number of transitions / number of states) except for the bddbddb-based tool whose number of transitions is unknown. The benchmarks were done on different computers but with very similar characteristics: dual core computers with 4GByte RAM. Comparison of the Tom-based and Bddbddb-based implementation can only done on the NSPK, View-Only and Combinatory examples because these are the only examples present in both papers.

The *Combinatory* example is a tiny TRS whose completion produces a large number of substitutions. Let $\mathcal{R} = \{g(f(x_1), h(h(h(x_2, x_3), x_4), x_5)) \to u(x_1, x_2, x_3, x_4, x_5)\}$ and $\mathcal{A}$ be the tree automaton whose transition set is the following: $\{nil \to q_h, f(q_{a_1}) \to q_f, g(q_f, q_h) \to q_g\} \cup \{t \to q_t, h(q_h, q_t) \to q_h \mid t \in \{a_i, b_i, c_i, d_i \mid i = 1, \ldots, 5\}\}$. For the variables x_1, x_3, x_4 and x_5 there are twenty possible instantiations during the completion. The variables x_1 and x_2 take only and respectively the values q_{a_1} and q_h. So, there are 20^3 transitions to compute by completion.

The *NSPK* example, i.e. Needham-Schroeder public key protocol is partially described in Section 5.1.2. This is, in fact, the version of the protocol corrected by G. Lowe (Lowe,

	Combinatory	NSPK	View-Only	Java prog. 1	Java prog. 2
TRS size (nb of rules)	1	13	15	279	303
Initial Automaton size	43 / 23	14 / 4	21 / 18	26 / 49	33 / 33
Timbuk 2.2:					
Final Automaton size	8043 / 23	151 / 16	730 / 74	1127 / 334	751 / 335
Time (secs)	**51.1**	**19.7**	**6420**	**25266**	**37387**
Timbuk 3.0:					
Final Automaton size	8043 / 23	259 / 104	353 / 100		
Time (secs)	**60.1**	**3.1**	**2452**		
Tom-based tool:					
Final Automaton size	8043 / 23	171 / 21	938 / 89	1974 / 637	1611 / 672
Time (secs)	**5.9**	**5.9**	**150**	**360**	**303**
Bddbddb-based tool:					
Final Automaton size	? / 25	? / 183	? / 97		
Time (secs)	**0.008**	**2.9**	**3.3**		

Figure 4.5: Benchmarks on three completion implementations

1996b) and used in (Genet and Klay, 2000). The *View-Only* example is another cryptographic protocol composing the *Smartright* system SmartRight (2001) designed by Thomson, see Section 5.1.3. Finally, *Java Prog* examples are TRS obtained by translation of Java bytecode applications as detailed in Section 5.2.

From this table, it is clear that the Bddbddb-based tool by (Gallagher and Rosendahl, 2008) is the most efficient one. What needs to be remarked, however, is that it does not tackle the same objective. Indeed, the approximation it computes is fully automatic and built-in the tool. As a consequence, if the approximation is too coarse, there is no way to refine it so as to prove the property. It is the case, for instance, for the NSPK and View-Only examples where the approximation computed by the Bddbddb-tool does not permit to prove the expected property whereas approximation computed by Timbuk and the Tom-based tool do. On all results computed by Timbuk 2.2 the expected properties were proved. It is also the case for Timbuk 3.0 except for the View-only example where only one of the numerous security has been proved (secrecy of so-called control words) using approximation equations.

4.6.3 Completion as an alternative for breadth-first search

For "positive" reachability analysis, i.e. proving that $s \rightarrow_{\mathcal{R}}{}^{*} t$, rewrite tools like Maude (Clavel et al., 2009) or Tom (Tom, 2009) generally give the fastest answer. For instance the `search` command in Maude is able to solve this problem even for non terminating TRSs. Given an initial problem of the form `reach s =>* t`, Maude rewrites s by $\mathcal{R}$ using a breadth-first search strategy and tables every obtained term. Tabling is used to avoid looping rewriting on a term. If a term u is in the table and if u is encountered again during rewriting then u will not be rewritten. If the term t is obtained then it is signaled by Maude.

Timbuk is able to achieve such "positive" proofs on left-linear TRSs. First, recall that using the exact normalization strategy (Definition 113 of Section 3.3.1) the produced abstraction function is injective. Then, using Corollary 77, we get that if $\mathcal{R}$ is left-linear,

$\mathcal{L}(\mathcal{A}^0) = \{s\}$ and there exists $n \in \mathbb{N}$ such that $t \in \mathcal{L}(\mathcal{A}^n_{\mathcal{R},\alpha})$ then $s \rightarrow^*_{\mathcal{R}} t$. Besides to this, we can remark that the completion algorithm is, by definition, also tabling all reachable terms. Each term is stored in the completed tree automaton structure. Furthermore, the tabling operation is even more advanced in Timbuk than in Maude since, because of normalization of tree automata transitions, every *subterm* of every reachable term is tabled. We can also note that the exact normalization strategy normalize each new term in a deterministic way, i.e. each term is tabled once. This guarantees a tabling strategy with maximal sharing. We here give an example illustrating all those aspects.

Example 135 *Let $\mathcal{R} = \{a \rightarrow f(a,b), b \rightarrow f(b,a)\}$ and the reachability problem be $a \rightarrow_{\mathcal{R}}{}^* f(f(a,b), f(b,a))$. Let $\mathcal{A}^0$ be the initial automaton with $\mathcal{Q}^0_f = \{q_0\}$ and $\Delta_0 = \{a \rightarrow q_0\}$. We now consider, on one side, what would be the table constructed by Maude and what are the completed automaton produced by* Timbuk.

0	*a*
1	*f(a,b)*
2	*f(f(a,b),b)* *f(a,f(b,a))*
3	*f(f(f(a,b),b),b)* *f(f(a,f(b,a)),b)* *f(f(a,b),f(b,a))* *f(a,f(f(b,a),a))* *f(a,f(b,f(a,b)))*
4	...

$\mathcal{A}^0$	$a \rightarrow q_0$
$\mathcal{A}^1_{\mathcal{R}}$	$f(q_0, q_1) \rightarrow q_2$ $b \rightarrow q_1$ $q_2 \rightarrow q_0$
$\mathcal{A}^2_{\mathcal{R}}$	$f(q_1, q_0) \rightarrow q_3$ $q_3 \rightarrow q_1$

As we can see, in both cases, after 3 rewritings on the Maude side and after 2 completion steps on Timbuk side, we can prove that $a \rightarrow_{\mathcal{R}}{}^* f(f(a,b), f(b,a))$. On the Timbuk side we have $f(f(a,b), f(b,a)) \rightarrow^*_{\mathcal{A}^2_{\mathcal{R}}} f(f(q_0,q_1), f(q_1,q_0)) \rightarrow^*_{\mathcal{A}^2_{\mathcal{R}}} f(q_2,q_3) \rightarrow^*_{\mathcal{A}^2_{\mathcal{R}}} f(q_0,q_1) \rightarrow^*_{\mathcal{A}^2_{\mathcal{R}}} q_0$ and q_0 is a final state of $\mathcal{A}^2_{\mathcal{R}}$. We thus have similar results. However, on Timbuk side we can see that the term a occurs only once, it is then replaced by the state q_0 which can be seen as its tabling index. Moreover, all terms that can be reached from a will also be recognized by q_0. In other words, q_0 represents all reachable terms from a. Hence, with regards to what Maude is doing, in the term $f(a,b)$ it is not necessary to explicitly consider the possible rewritings of a into $f(a,b), f(f(a,b),b), \ldots$ since in the tree automaton a has been replaced by q_0 in the term $f(a,b)$ and q_0 will recognize all terms reachable from a. Finally, note that the above example is in the **G** class of Section 2.1.1 and that completion *ends* on the $\mathcal{A}^2_{\mathcal{R}}$ tree automaton, i.e. every possible reachable by rewriting a with $\mathcal{R}$ is recognized by $\mathcal{A}^2_{\mathcal{R}}$.

In spite of the sophisticated tree automata data structure of Timbuk, Maude is at least 10 times faster than Timbuk to prove reachability on examples where the TRS has a narrow rewriting tree. This difference is less important on TRSs which are more combinatorial and, thus, having a wider rewriting tree. For instance on the TRS encoding the readers-writers protocol of Section 4.6.4, which is only a bit more non-deterministic, Timbuk is only 3 times slower for reachability problems on big terms. Finally, we also experimented with TRSs automatically generated from Java byte code programs with Threads (see Section 5.2). Those TRS are huge and manipulate big terms. For instance, the following Java

program can be compiled into bytecode and then into a TRS of around 2000 rules.

```
class T extends java.lang.Thread{
    Object lock;
    public T(Object lock){
        this.lock = lock;
    }

    public void action(){
        System.out.println(1);
    }

    public void run(){
        synchronized(lock){
            this.action();
        }
    }
}

class ExThreads1{
    public static void main(String[] argv){
        Object lock = new Object();
        T thread = new T(lock);
        T thread2 = new T(lock);
        thread.start();
        thread2.start();
        try{
            thread.join();
        } catch (InterruptedException e){}
        System.out.println(2);
        System.out.println(3);
    }
}
```

Using the Copster tool (see Section 5.2), we can export both Timbuk and Maude specifications. In our encoding of Java programs into TRS, outputs are lists of terms stored as a parameter of the term representing the Java program state. On the above example, we can check that any output list $[2, 3, 1, 1]$ is unreachable. This can be automatically proved using Timbuk and the exact completion strategy in less than a minute and 300 Mb of memory. On the same example Maude is not able to answer: it has to be stopped since it exhausted the memory (1 Gb) in less than 3 minutes. Moreover, the fixpoint automaton, computed by Timbuk and used to prove unreachability, was certified using the checker (of Section 4.5) within seconds.

4.6.4 Equational abstraction with Maude

We borrow the example of readers-writers system from Section 9.3 of (Clavel et al., 2008). States are represented by terms of the form `state(R,W)` where `R` indicates the number of readers and `W` indicates the number of writers accessing the critical resource. Readers

and writers can leave the resource at any time. A writer can gain access to the resource if nobody else is using it, and readers only if no writer is using it. The initial state is `state(o,o)` where `o` is the term representing the number 0. The two properties to prove are *mutual exclusion*, i.e. readers and writers never access the resource at the same time and *one writer*, i.e. at most one writer accesses the resource at the same time. This is defined by the following Timbuk specification.

```
Ops state:2 o:0 s:1
Vars R W
TRS R1
    state(o,o) -> state(o,s(o))    % Writer can start if alone
    state(R,o) -> state(s(R),o)    % Reader can start if no writer
    state(R,s(W)) -> state(R,W)    % Readers and writers can stop
    state(s(R),W) -> state(R,W)    % anytime
Set A1
    state(o,o)
Patterns
    state(s(_),s(_))     % at least a writer and a reader at the same time
    state(_,s(s(_)))     % at least two writers
Equations Abs
Rules
    s(s(_))=s(s(o))
```

In Timbuk 3.0 specifications, the `TRS` section is followed by either an `Automaton` or `Set` section defining the initial set of terms, a `Patterns` section defining the patterns of forbidden terms and an `Equation` section defining the approximation equations. When the set of initial terms has an unbounded size, it is given using a tree automaton. In the previous specification, since there is only one initial term, it is thus given as a finite set of ground terms. Patterns are terms with variables (possibly anonymous, i.e. the '_' used here) that are matched on the completed tree automaton. If a substitution is found then a ground term matched by the pattern is reachable. Finally, in the equation section we can find the approximation equation used here. Note that it is defined using an anonymous variable and is thus noted `s(s(_))=s(s(o))` but could also be written `s(s(R))=s(s(o))`. This equation merges every term representing a natural number $x \geq 2$ with 2. Processing this specification using Timbuk we obtain immediately a fixpoint tree automaton where no occurrence of the forbidden patterns is found. This means that the mutual exclusion and one-writer properties are guaranteed on this specification. Note that, here, no other proof are needed to prove those properties. This is different from (Clavel et al., 2008) where some additional properties have to be proven on the oriented set of approximation equations: ground confluence, sort-decrease, termination, sufficient completeness and ground coherence.

Furthermore, since the initial set of terms is finite, we can easily build a $\mathcal{R}/E$-coherent tree automaton recognizing it. For instance, if t=`state(o,o)` is the term to recognize, the easiest way to do that is to normalize t using new states and the $Norm_{\Delta}$ function (Definition 99). In the resulting tree automaton $\mathcal{A}$, since every state recognize exactly one term, $\mathcal{A}$ is trivially $\mathcal{R}/E$-coherent. Thus, from Theorem 110, we get the the guarantee that the fixpoint recognize only terms of $\mathcal{R}^*_E(\{t\})$.

The equation `s(s(_))=s(s(o))` is enough to prove the properties but it is simpler than the one used in (Clavel et al., 2008). In (Clavel et al., 2008) the equation is more precise since it states that only natural numbers for readers should be abstracted. This approximation is enough since the system should only give access to at most one writer at a time. Hence, no approximation is necessary on the writer number. For defining similar approximations, in Timbuk we use *contextual equations* (see Section 4.1.2) of $\mathcal{A}$. In our specification, we can thus replace the equation `s(s(_))=s(s(o))` by the following contextual equation:

```
[state(s(s(R)), o)] => [s(s(R))=s(s(o))]
```

meaning that every natural number $x \geq 2$ is merged with 2 provided that it appears on the reader position. This leads to a more precise approximation comparable to the one used in (Clavel et al., 2008).

The second example is the bakery protocol of (Meseguer et al., 2003). In this protocol two processes are accessing a common critical section. Each process has a state that is either `sleep`, `wait` or `crit` and a ticket value, which is a natural number. Initially each process is in `wait` state with a `o` value for its ticket. When a process wants to access to the critical section, he sets its ticket to the ticket value of the other process plus one and moves to state `wait`. A waiting process can move to the `crit` state if the ticket value of the other process is `o` or if he has the smallest ticket. Then, when leaving the critical section the process goes back to `sleep` state and sets its ticket to `o`. The safety property to prove is, of course, that the two processes cannot be in `crit` state at the same time. The original Maude specification is a conditional TRS where tickets are compared using the '$<$' Maude built-in operator. This is encoded using Timbuk's restricted form of conditional rules where conditions can be put on variables. Here is a possible Timbuk specification for the bakery protocol with two processes.

```
Ops state:4 o:0 s:1 sleep:0 wait:0 crit:0
Vars X Y P Q Z U
TRS R
    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,_)
```

With this TRS $\mathcal{R}$ the objective is thus to prove that $\forall i, j \in \mathbb{N} : state(sleep, 0, sleep, 0) \not\rightarrow_{\mathcal{R}}^{*} state(crit, i, crit, j)$. To prove this property, an abstraction is needed because $\mathcal{R}$ is not ter-

minating and the values for i and j are not bounded. Here is a possible execution illustrating this last point.

$$\begin{aligned}
state(sleep, 0, sleep, 0) &\rightarrow_{\mathcal{R}} state(wait, s(0), sleep, 0)\\
&\rightarrow_{\mathcal{R}} state(crit, s(0), sleep, 0)\\
&\rightarrow_{\mathcal{R}} state(crit, s(0), wait, s(s(0)))\\
&\rightarrow_{\mathcal{R}} state(sleep, 0, wait, s(s(0)))\\
&\rightarrow_{\mathcal{R}} state(wait, s(s(s(0))), wait, s(s(0)))\\
&\rightarrow_{\mathcal{R}} state(wait, s(s(s(0))), crit, s(s(0)))\\
&\rightarrow_{\mathcal{R}} state(wait, s(s(s(0))), sleep, 0)\\
&\rightarrow_{\mathcal{R}} state(wait, s(s(s(0))), wait, s(s(s(s(0)))))\\
&\rightarrow_{\mathcal{R}} \ldots
\end{aligned}$$

A possible abstraction is to define equivalence classes between states. In the above rewriting derivation, we can remark that the term $state(wait, s(0), sleep, 0)$ and the term $state(wait, s(s(s(0))), sleep, 0)$ play a similar role. In fact, to prove the safety, only three distinct tickets value need to be considered: 0, 1 and 2. The 0 value represents only tickets of value 0. For any state of the form $state(x, i, y, j)$, if $i = 0$ and $j \neq 0$ then j is mapped to any value different from 0, say 1. If $i \neq 0$, $j \neq 0$ and $i > j$ then i is mapped to 2 and j to 1, and vice versa. This is the abstraction used in (Meseguer et al., 2003). In (Meseguer et al., 2008), the definition of the abstraction and the verification process are more precisely detailed. The abstraction is defined using 6 equations, 4 being conditional. Conditional equations are necessary in this case to map i to 2 and j to 1 *only if* $i > j$. As said in the paper, those equations are more complex than required by the abstraction principle so as to get a terminating set of oriented equations. The paper also details the necessary proofs of ground convergence of the oriented equations and ground coherence of the TRS and oriented equations.

In Timbuk, since we do not have comparable conditional equations, it is not possible to define the same approximation. However, by a careful observation of the above rewriting derivation, one may remark that values of i and j are linked. If one of the tickets is i then the other ticket j is either 0, $i+1$ or $i-1$. This is why another abstraction is possible. It simply needs to distinguish between 0, i, $i-1$ and $i+1$. This can be done by using 4 distinct equivalence classes: $[0], [1], [2], [3]$. Like above, the $[0]$ class contains only 0, $[1]$ contains tickets $\{1, 4, 7, \ldots\}$, $[2]$ contains tickets $\{2, 5, 8, \ldots\}$ and $[3]$ contains $\{3, 6, 9, \ldots\}$. This abstraction permits to prove the property because, for any two tickets i, j and term $state(x, i, y, j)$ where $i \geq 1$, it is enough to distinguish between $j = 0$, $j = i - 1$ and $j = i + 1$. For instance, assume that $i = 3$ and $j = 4$ then $[i] = [3]$ and $[j] = [1]$ which are distinct classes. Furthermore, since we have $s(i) = j$ we also have $s([i]) = [j]$, i.e. $s([3]) = [1]$. Hence, the process having the ticket 3 is guaranteed to enter critical section *before* the process having ticket 4[2]. Note that we need at least the above equivalence classes. Restricting to a smaller set of equivalence classes $[0] = \{0\}, [1] = \{1, 3, 5, \ldots\}, [2] = \{2, 4, 6, \ldots\}$ would merge too many terms and prevent us to prove the property. With such equivalence classes, if a process p_1 has the ticket 2 and another one p_2 has the ticket 3, p_1 can enter the critical

[2]This is also due to the fact that the TRS do not use conditions on $<$ but direct comparisons with regards to the number of $s(\ldots)$ symbols.

section. But, since $[3] = 1$ and $s([3]) = s([1]) = 2$ process p_2 can do the same, leading to a safety violation.

Let us come back to the good approximation based on 4 equivalence classes $[0] = \{0\}, [1] = \{1, 4, 7, \ldots\}, [2] = \{2, 5, 8, \ldots\}, [3] = \{3, 6, 9, \ldots\}$. An interesting point, here, is that this abstraction can be done in Timbuk with the single equation $E = \{s(s(s(s(X)))) = s(X)\}$. With this equation, completion terminates and Timbuk can prove the safety property on the over-approximation of reachable terms. Note that, the above proof can be carried out in Timbuk only because of the specific encoding of '$<$' using restricted conditional rules. This encoding relies on the fact that with two processes, for any two tickets i and j, $i < j$ if and only if $j = i + 1$. However, a similar remark holds for any number of processes. For instance for 3 processes having three tickets i, j, k, the process having the ticket i can enter in critical section if either $j = 0$ and $k = 0$, $j = 0$ and $k = i+1$, $j = i+1$ and $k = 0$, $j = i + 1$ and $k = i + 2$, or $j = i + 2$ and $k = i + 1$. Hence, we can encode the bakery protocol for any fixed number of processes n in a Timbuk TRS in a similar way. As a result, since 3 equivalence classes plus one for 0 were necessary to abstract 2 processes, $n + 1$ classes plus one for 0 should be necessary to abstract n. Hence, for n processes an equation of the form $s(s^{n+1}(X)) = s(X)\}$ should be enough to prove the safety property.

To conclude on this part, by restricting to safety properties, conditional left-linear TRS and linear equations, we do not not need any additional proof of ground convergence, coherence, etc. to guarantee the safety of the system. This is a real advantage w.r.t. the equational abstractions defined by (Meseguer et al., 2003). One of the main advantage is that, except linearity, no restriction is made on the set of equations. However, their framework is more general and permits to tackle liveness properties which are, for the moment, out of reach of the equational completion and the Timbuk tool. Note that another restriction they have, finite sets of initial terms, have even recently been weakened using narrowing in (Escobar and Meseguer, 2007).

Chapter 5

Applications

This chapter reviews two of the main applications of tree automata completion to verification: security proof of cryptographic protocol and Java bytecode analysis. Those two problems are of a very different nature and this is intentional. The first objective is to show that both the modeling using TRS and the verification using tree automata completion are agile techniques. The second objective is to demonstrate that the technique scales to verify an industrial cryptographic protocol and *large* term rewriting systems generated by translation of Java bytecode into TRS. Finally, in the last section, we briefly present another application of tree automata completion to prove deadlock freeness of programs.

5.1 Cryptographic protocol verification

5.1.1 The problem

A cryptographic protocol is a protocol executed by several agents through a network where the messages, or at least some parts of the messages, are produced using cryptographic functions (encryption, hashing, ...). In the following we denote by $\langle x, y\rangle$ the pair of message components x and y and by $\{x\}_k$ the result of the encryption of the message x by the key k. Note that, we use the same notation for symmetric and asymmetric encryptions. For any key k, the inverse key will be denoted by k^{-1}. For symmetric encryption, we will assume that $k = k^{-1}$. Finally the result of the hashing of a message x is denoted by $hash(x)$ and the Xor of two messages x and y is denoted by $Xor(x, y)$. For readability, $\{\langle x, y\rangle\}_k$ is noted $\{x, y\}_k$ and $\langle x, \langle y, z\rangle\rangle$ is noted $\langle x, y, z\rangle$, i.e. we assume that $\langle , \rangle$ is associative. Cryptographic protocols are used for various purposes between the agents:

- exchanging secret information (like secret keys)
- authenticating one, some or all the agents
- achieving a transaction (Electronic Commerce)
- voting

- ...

The mechanism of the protocol is supposed to resist to the attacks of, so called, intruders who are dishonest agents. Verifying a protocol consists in checking that whatever the intruders activity may be they should not be able to modify the expected result of the protocol, e.g.

- obtain a secret information
- pretend to be someone else
- modify a transaction
- vote several times

Although the core of a cryptographic protocol is usually small [1], its formal verification is hard. Indeed, for the verification to be relevant, it is necessary to take intruders into account and this leads to very combinatorial models. One of the most famous intruder model is the one by Dolev and Yao (D.Dolev and Yao, 1983). This model is informally summarized by two mottos: "cryptographic functions are blackbox" and "the intruder is the network". The first motto means that cryptographic functions (symmetric, asymmetric, hash functions) are supposed to be invulnerable. For instance, this model assumes that the intruder cannot obtain any information from $\{m\}_k$ nor modify it without knowing k. This assumption is very strong and not always true in practice since it depends on the robustness of the cryptographic function used, on the length of k, on the fact that m has to be unpredictable etc. However, by careful choices during the implementation of m, k and $\{m\}_k$, such assumption is realistic. On the opposite, the second motto "the intruder is the network" is an assumption that may be stronger than reality. This means that the intruder is able to

- intercept, block, replay any message,
- store any message or message component he has seen,
- decrypt a message if he knows (has stored) the inverse key,
- encrypt any message with any key he knows,
- compose or decompose message he knows.

This can be summarized by the set of deduction rules of Figure 5.1, following (Clarke et al., 1998). These rules perform deduction on S the, so-called, knowledge or store of the intruder. The store represents the set of informations the intruder can intercept during the execution of the protocol.

Besides to this, when protocols are designed for use on open networks, like Internet protocols for instance, the verification on Dolev-Yao model should hold for an unbounded number of agents executing the protocol at the same time and as often as needed. The verification has, thus, three unbounded dimensions: the number of agents running the protocol,

[1] It generally consists of 3 to 6 exchanged messages.

$$(Axiom)\ \frac{}{S, M \vdash M} \qquad (Decryption)\ \frac{S \vdash \{M\}_K \quad S \vdash K^{-1}}{S \vdash M} \qquad (Encryption)\ \frac{S \vdash M \quad S \vdash K}{S \vdash \{M\}_K}$$

$$(UnpariringL)\ \frac{S \vdash \langle X, Y\rangle}{S \vdash X} \qquad (UnpairingR)\ \frac{S \vdash \langle X, Y\rangle}{S \vdash Y} \qquad (Pairing)\ \frac{S \vdash X \quad S \vdash Y}{S \vdash \langle X, Y\rangle}$$

Figure 5.1: Dolev-Yao intruder capabilities as deduction rules

the number of sessions run by each agent and the number of intruder's action. As a result, if the verification is performed using a model-checker, it will only be on a bounded abstraction of the original problem (Lowe, 1996a). Note that some works propose model-checking techniques avoiding the bound on the number of intruder's action like (Turuani, 2003).

5.1.2 Encoding into TRS and verification by tree automata completion

TRS are a convenient formal way to model cryptographic protocols. As far as we know the first work using TRS for this purpose is (Denker et al., 1998) for attack detection. Then, it was followed by many other works like (Jacquemard et al., 2000) and (Genet and Klay, 2000) where the rewriting models are used for security proof. Those works use a set of rewrite rules for the intruder close to deduction rules of Figure 5.1. Encoding those deduction rules into a TRS needs to overcome a technical detail: deduction rules operate on sets and rewrite rules on terms. Though sets are difficult to encode as terms, multisets are enough and can be encoded into terms using an associative and commutative symbol. This is the case, for instance, in (Genet and Klay, 2000) with the symbol $\sqcup$[2] and also in (Jacquemard et al., 2000). The encoding is even clearer in (Rusinowitch and Turuani, 2001). The set of deduction rules of 5.1 can thus be encoded by the following set of rewrite rules where $\sqcup$ is supposed to be an associative and commutative operator. Those rules rewrite a term (a multiset) representing the knowledge of the intruder.

$(Decryption)\ \{M\}_K \sqcup K^{-1} \rightarrow M \sqcup \{M\}_K \sqcup K^{-1}$

$(Encryption)\ M \sqcup K \rightarrow \{M\}_K \sqcup M \sqcup K$

$(UnpairingL)\ \langle X, Y\rangle \rightarrow X \sqcup \langle X, Y\rangle$

$(UnpairingR)\ \langle X, Y\rangle \rightarrow Y \sqcup \langle X, Y\rangle$

$(Pairing)\ X \sqcup Y \rightarrow \langle X, Y\rangle \sqcup X \sqcup Y$

This is for the representation of the Dolev-Yao intruder with TRS. For the verification part, the aim is to finitely represent the infinite set of messages that an intruder is able to construct. For that purpose, tree automata are now a very commonly used technique. The idea of using tree automata for verifying those specific softwares was independently proposed by (Monniaux, 1999; Genet and Klay, 2000; Goubault-Larrecq, 2000). The common intuition behind those works is the following: represent the potentially infinite knowledge of the intruder as a tree automaton. (Monniaux, 1999) encodes the disassembly of the messages using TRS. The concerned TRS is thus restricted to the rules:

[2]Note that the associative and commutative behavior is made explicit in (Genet and Viet Triem Tong, 2001).

$(Decryption)\ \{M\}_K \sqcup K^{-1} \rightarrow M \sqcup \{M\}_K \sqcup K^{-1}$

$(UnpairingL)\ \langle X, Y\rangle \rightarrow X \sqcup \langle X, Y\rangle$

$(UnpairingR)\ \langle X, Y\rangle \rightarrow Y \sqcup \langle X, Y\rangle$

In (Monniaux, 1999), pairing and encryption operations of the intruder (the other rules) are simulated by the tree automaton itself. This requires a separated manual proof. However, in this case, the knowledge of the intruder can entirely be represented by a regular set of non associative and commutative terms, and the operator $\sqcup$ becomes useless. The TRS used in (Monniaux, 1999) uses explicit decryption and unpairing operators as follows:

$(Decryption)\ decrypt(\{x\}_k, k^{-1}) \rightarrow x$

$(UnpairingL)\ proj1(\langle x, y\rangle) \rightarrow x$

$(UnpairingR)\ proj2(\langle x, y\rangle) \rightarrow y$

The set of reachable terms w.r.t. to this TRS is over-approximated using a tree automaton. However, though it is not mentioned by Monniaux, the above TRS is in the **RL-M** class and reachable terms can thus be exactly computed.

The technique proposed in (Genet and Klay, 2000) and (Goubault-Larrecq, 2000) goes further and represent the entire protocol execution and intruder activity using a TRS. In this case, the corresponding TRS is no longer in a regular class of Section 2.1.1 and the set of reachable terms needs to be approximated. These two works use a tree automata completion algorithm close to the one presented in Section 3.1 to perform the verification.

In (Genet and Klay, 2000), the case study was the Needham-Schroeder public key protocol (Needham and Schroeder, 1978) (NSPK for short) which was, at that time, the typical benchmark of security protocol verification techniques. On this protocol, once the normalization rules are defined, secrecy can be verified automatically on an unbounded number of agents, protocol sessions and intruder's basic actions. As far as we know, (Genet and Klay, 2000) was one of the first technique able to prove such a general result in a (semi-)automatic way. Before that, D. Bolignano (Bolignano, 1996) proposed to prove protocols using abstract interpretation, and thus approximations. However, proofs were essentially manual and done in the Coq proof assistant.

Another interest of tree automata based verification, that aroused during this experiment, is that it eases the minimization of models. For instance, on the NSPK example, if we achieve an exact completion after 4 steps the completed automaton has already more than 4500 states and is, of course, not complete. This shows the state explosion problem we face on such specifications. On the same initial tree automaton and TRS, with a very rough approximation, completion terminates after 6 steps and obtain a fixpoint automaton with only 16 states. In such an approximation, for instance, we even forget the message structure itself, i.e. the ordering of components in the messages is lost. Although this fixpoint is imprecise, it is still sufficient to prove the security property. The benefit of this experiment was thus to show that the fine tuning of approximations is a good way to scale-up verification by tree automata completion to real-size problem. This is illustrated in Section 5.1.3 on an industrial cryptographic protocol and in Section 5.2 on Java bytecode verification.

5.1.3 The SmartRight case study

The *SmartRight* system

The *SmartRight* system (SmartRight, 2001) offers a high protection against illegal copies within a digital home network. The *SmartRight* system was proposed to the Digital Video Broadcasting (DVB) ad-hoc group (DVB, 2003) on Copy Protection Technology in reply to a call for proposals on Copy Protection and Content Management technologies. The security is vital to the *SmartRight* system, since the subvert of system may cause serious revenue loss of content industry. The security of the *SmartRight* system relies on the two key technologies:

- Cryptography: the system uses a number of cryptographic protocols to create the personal private network and protect content against illegal copies.
- Smart cards: tamper resistance of smart cards offers a secure environment to handle secret keys needed for the protocols. Furthermore, it guarantees a trusted behavior of main elements of the system.

The design of a cryptographic protocol is a particularly error-prone process. In addition to the choice of well-designed cryptographic functions, we must ensure that the cryptographic protocols do not have flaw. This is the reason why Thomson R&D was interested in using formal methods to verify the cryptographic protocols in the *SmartRight* system.

	Converter Card	Communication	Terminal Card
1.	get new CW generate random $VoKey$, VoR		
2.		$\{VoKey, Xor(CW, VoR)\}_{K_c} \longrightarrow$	
3.			decrypt using K_c extract $VoKey$, $Xor(CW, VoR)$ generate random $VoRi$
4.		$\longleftarrow VoRi$	
5.	delete $VoKey$, VoR	$VoR, \{Hash(VoRi)\}_{VoKey} \longrightarrow$	
6.			verify $\{Hash(VoRi)\}_{VoKey}$ using $VoKey$, then extract CW using VoR

Figure 5.2: specification of the protocol

The 'view-only' protocol is one of the cryptographic protocols used in *SmartRight*. Figure 5.2 shows a complete execution of the protocol (called a session). This protocol is deployed between an access device (e.g. a decoder) receiving a scrambled digital content (e.g. video) and a presentation device (e.g. television) which is supposed to unscramble the content before rendering it. The keys used to scramble the content are called *control words* (*CW*) and change periodically (typically every 10 seconds). Both access device and presentation device are equipped with a smartcard, called Converter Card (CC) and Terminal Card (TC), respectively. CC and TC share a secret symmetric encryption key Kc which is embedded in the card. The access device encrypts *CW* and sends it together with the scrambled content to the presentation device (step 2). The presentation device extracts CW (step 6) and unscrambles the content if it received a good response (step 5) to the challenge (step 4).

The property to prove is the following: the control word *CW* (input of CC at step 1) may be extracted by TC (at step 6.) only once at the time where the protocol is played. This property guarantees that the content can be viewed only once. Even if the content (encrypted CW + scrambled content) is recorded, the protocol ensures that the sequence of *CW* cannot be replayed. This property has to be verified under the following assumptions: a hacker can impersonate the Converter Card and the Terminal Card and it can attack all exchanged messages of the protocol: modify, delete a message during a session or replay a message of a session in following sessions. However, it is impossible for him to know K_c or to modify the behavior of CC and TC since they are sealed in the smart card.

Using Timbuk for verifying SmartRight

This protocol runs over a home network and we can thus limit the analysis to a bounded number of agents. On the opposite, the verification must hold for any number of protocol sessions. Thus, this verification cannot be performed using a model-checking tool which only inspect a finite number of sessions. A session is a sequence of protocol configurations. A configuration of the protocol is a record of actors' and intruder's knowledge as well as the exchanged messages. Each configuration is represented by a term. A step of the protocol is described using a rule rewriting a configuration to another. Thus, executing the protocol consists in rewriting repeatedly a term representing the current configuration of the system. Intruder's actions are also modeled using rewrite rules.

Starting from an initial configuration t and using Timbuk, we compute a tree automata recognizing an over-approximation of all the configurations that can be reached by applying the TRS (finitely or not) on t. Then, to verify a property on the protocol, it is enough to represent faulty configurations by terms and check that they are not recognized by the automaton.

Modeling the *SmartRight* protocol

Both the Converter Card and the Terminal Card are represented by a term containing their respective knowledge. The term $CC(l, v_1, v_2)$ will represent a Converter Card whose list of incoming control words is l and whose current *VoKey* is v_1 and current *VoR* is v_2. Similarly the Terminal Card is represented by a term $TC(i, v, l)$ where v is the current *Vori*, l is a term of the form the $Read(CW_n, \ldots Read(CW_1, nil))$ where $CW_1, \ldots, CW_n$ are control words that have been accepted by the Terminal Card, in this order. In i is stored the last pair ($VoKey$, $Xor(CW, VoR)$) received by the Terminal Card. Keys *VoKey*, *VoR* (resp. *VoRi*) must be fresh for each new session of the Converter Card (resp. Terminal Card).

A safe modeling choice for a key generated by a card c is to make it depend on an information known by c and that is changing at the beginning of every new session of c. For instance, the Converter Card starts a new session for every new control word cw it receives. Thus we represent randomly generated keys of the Converter Card by the terms $VoKey(cw)$ and $VoR(cw)$. On the other hand, when the Terminal Card generates $VoRi$ it does not have access to cw so it is not possible to make depend $VoRi$ on this parameter. A safe choice is to build this key as follows: $VoRi(v, l)$[3] where v is the last $VoKey$ the

[3]Note that $VoRi(v)$ would not be fresh enough since if the intruder replays the first message of the Converter

Terminal Card received and l is its list of accepted control words. The two smart cards are supposed to communicate in an insecure world: the network is supposed to be ruled by a Dolev-Yao intruder. In the model, the intruder is represented by its knowledge: a term of the form $Store(e_1, \ldots Store(e_n, emptystore))$ where $e_1, \ldots, e_n$ are the pieces of messages the intruder knows. Finally a configuration in *SmartRight* is represented by a term of the form $State(m, cc, tc, st)$ where m is the current message, cc, tc and st are respectively the Converter Card, the terminal card and the intruder.

Steps of the protocols are described with rewrite rules. We first show the rule that initializes the protocol session: the converter card receives some control word and sends the first message (all small letters denote variables and $x :: l$ denotes the list whose first element is x and whose remainder is l).

$$State(m, CC(cw :: l, v_1, v_2), tc, i) \rightarrow$$
$$State(\{VoKey(cw), Xor(cw, VoR(cw))\}_{K_c}), CC(cw :: l, VoKey(cw), VoR(cw)), tc, i)$$

This rule corresponds to the steps (1) to (2) of the protocol where the converter card has at least one control-word cw to send. The other steps of the protocol are encoded in the same way.

$$\text{(2) to (4)} \quad State(\{vok, e\}_{K_c}, cc, TC(x, y, cwl), i) \rightarrow$$
$$State(vori(vok, cwl), cc, TC(\langle vok, e\rangle, vori(vok, cwl), cwl), i)$$

$$\text{(4) to (5)} \quad State(vor, CC(cw :: l, run, v_1, v_2), tc, i) \rightarrow$$
$$State(\langle vor, \{v_1, hash(vor)\}_{K_c}\rangle, CC(l, init, v_1, v_2), tc, i))$$

$$\text{(5) to (6)} \quad State(\langle vor, \{vok, hash(vori)\}_{K_c}\rangle, cc, TC(\langle vok, Xor(cw, vor)\rangle, vori, cwl), i) \rightarrow$$
$$State(nil, cc, TC(nil, nil, read(cw, cwl)), i)$$

Besides to this, the intruder's actions are encoded by another set of rewrite rules. Here are the rewrite rules for the intruder:

- The intruder can read and store every sent message and he can send every data he knows:

  $State(x, cc, tc, st) \rightarrow State(x, cc, tc, Store(x, st))$

  $State(y, cc, tc, Store(x, st)) \rightarrow State(x, cc, tc, Store(x, st))$

- The intruder can flip elements in its store (he can reorder it):

  $Store(x, Store(y, z)) \rightarrow Store(y, Store(x, z))$

Card with a key v and a control word cw, then the reply sent by the Terminal Card will be identical: $VoRi(v)$, so will be the third message and we will get an attack: cw will be read twice.

- The intruder can decrypt a message m encrypted with k if he has k (k is a symmetric key):
  $Store(\{m\}_k, Store(k, st)) \rightarrow Store(m, Store(encrypt(k, m), Store(k, st)))$
- An intruder can crypt any element of its store with key (or anything else) of its store:
  $Store(x, Store(k, st)) \rightarrow Store(encrypt(k, x), Store(x, Store(k, st)))$
- An intruder can decompose message that are not encoded:
  $Store(\langle x, y\rangle, st) \rightarrow Store(x, Store(y, Store(\langle x, y\rangle, st)))$
- An intruder can compose messages from elements found in the store:
  $Store(x, Store(y, st)) \rightarrow Store(\langle x, y\rangle, Store(x, Store(y, st)))$
- The intruder can apply a hash function:
  $Store(x, st) \rightarrow Store(Hash(x), Store(x, st))$
- The intruder can build Xor:
  $Store(x, Store(y, st)) \rightarrow Store(Xor(x, y), Store(x, Store(y, st)))$
- The intruder can decompose Xor if he has one of the elements:
  $Store(Xor(x, y), Store(x, st)) \rightarrow Store(y, Store(Xor(x, y), Store(x), st)))$
  $Store(Xor(x, y), Store(y, st)) \rightarrow Store(x, Store(Xor(x, y), Store(y, st)))$

The verification of *SmartRight*

Adjusting the initial tree automaton makes it possible to analyze the protocol starting from different hypothesis or different initial configurations. It is possible to study what happens if the intruder knows some keys, or some control word by adding them to the initial configuration of the intruder. Similarly, we can study the behavior of the protocol for different capacity of the intruder by incrementally adding some rewrite rules to the intruder behavior.

Starting from the initial configuration and from the TRS modeling both the protocol and the intruder actions, we use Timbuk to compute the set of reachable terms. We used it both in exact and approximated mode. In practice, the exact mode helps in finding attacks and the approximated mode is used, when attacks are no longer found, to prove that there are no more. For the *SmartRight* protocol verification, our methodology for building approximation was the following:

"Observe one protocol session among a safe over-approximation of an unbounded number of past and future protocol sessions, all attacked by an unbounded number of intruder's actions". In the approximation we built, the list of control words received by the Converter Card is approximated by normalization rules into $[old_cw, \ldots, old_cw, current_cw, next_cw, \ldots, next_cw]$. In other words, all control words are approximated by only three equivalence classes. The $current_cw$ contains only one control word which is the observed one. All past (resp. next) control words are all considered as equivalent and merged into the class old_cw (resp. $next_cw$). Then, to check the *anti-replay*

property, it is enough to use Timbuk's pattern matching (see Section 4.1.2) on the approximation automaton and verify that a pattern of the form $Read(x, Read(y, z))$ does only accept solutions where old_cw, $current_cw$, $next_cw$ are accepted in this order and $current_cw$ at most once. For instance, one has to verify that no solution of the form $Read(old_cw, Read(current_cw, \ldots))$ is obtained (this solution represents an old control word that have been accepted after the current control word). For *SmartRight*, we have computed several tree automata recognizing the over-approximations of reachable configurations of the protocol under different hypothesis. First, we have verified the model of the protocol without any intruder action. This has permitted to validate the model by verifying that if the control words are *all* emitted by the Converter Card then some of them[4] are received, at most once and in the right order by the Terminal Card. Secondly, we computed an approximation with a stronger intruder able to listen, erase and replay messages. On the computed automaton, while searching for a pattern $Read(x, Read(y, z))$, we found:

```
Solutions:
[...]
solution 4: x <- qcurrent_cw, y <- qold_cw, z <- qnil,
solution 5: x <- qcurrent_cw, y <- qcurrent_cw, z <- qreadcurrent,
solution 6: x <- qcurrent_cw, y <- qcurrent_cw, z <- qreadold,
[...]
```

where $Read(current_cw, Read(current_cw, \ldots))$, shows that control words may be accepted repeatedly (but in the right order). A careful study of the automaton pointed out that this attack was due to an element missing in the protocol specification. After step 6. of the protocol, if the intruder replays the message $\{VoR, \{Hash(VoRi)\}_{VoKey}\}_{K_c}$ before the terminal card starts its new session and changes its key $VoRi$, then this message can be read several times by the Terminal Card and the control word may also be extracted several times. This is no longer possible if the terminal card deletes its key $VoRi$ as soon as it has extracted the control word. This is exactly what had been added to the Thomson specification to fix the problem. On this fixed specification the *anti-replay* property has been proved.

Finally, we have computed a third approximation automaton for an intruder having maximal capabilities w.r.t. the verification assumption. We have given to the intruder some keys (but not K_c) and the ability to listen, erase, replay, construct, deconstruct, decrypt (if he has the key), encrypt messages and also the ability to construct or deconstruct Xors. The completed automaton has 74 states and 732 transitions. On this approximation automaton, it was possible to prove the following properties:

Secrecy: K_c, Vo_Key and all the control words CW remain secret. This can simply be done by checking that no pattern of the form $Store(K_c, x)$, $Store(VoKey(x), y)$, etc. can be found in the automaton.

Authentication Control words sent by the Terminal Card are all correctly authenticated, i.e. no control word sent by the intruder is accepted. This can be done by checking

[4]They are not all received since a Converter Card may start a new session with a new control word before the previous session was completed. Note that, we also modeled the case where TC can complete its session before CC starts a new one and we proved a form of liveness where all the sent control words are received, at most once and in the right order.

that a pattern of the form $Read(x, y)$ accepts no solution mapping x to a data owned by the intruder.

Anti-replay As before, we can check that control words are received at most once and in the right order using a pattern of the form $Read(x, Read(y, z))$.

The *anti-replay* property proved on this protocol is stronger than classical secrecy and authentication properties and heavily relies on the freshness of the data sent in the messages. With Timbuk, we are able to prove this property in an automatic way with some user interactions limited to the definition of the normalization rules. As far as we know, this kind of property seems difficult to tackle, on an unbounded number of sessions, with a fully automatic verification tool (Catherine Meadows, 2001). It could have also been proved using a proof assistant like in (Paulson, 1997; Bolignano, 1996) since induction is well adapted to handle properties where freshness of keys and nonces is crucial. Nevertheless, we believe that using a proof assistant would have needed more time and energy to achieve the same goal.

Those experiments with the exact completion strategy of Timbuk permitted to detect very early some omissions in the Thomson specification. Finally, rewrite approximations permits to prove automatically the property in a general setting: unbounded number of sessions, unbounded number of intruder actions.

5.1.4 Comparison with other tools

The preliminary work on the Needham-Schroeder public key protocol was carried out in 2000 and the above case study on the SmartRight protocol was done in 2003. Then, this approach was automatized in (Oehl et al., 2003; Boichut et al., 2004) and integrated in the AVISPA security protocol verification tool (Armando et al., 2005). It was extended for attack reconstruction in (Nesi et al., 2003). Another extension was proposed in (Zunino and Degano, 2006) so as to deal with non linear rules of specific operators like the Xor.

However, Timbuk is not targeted to verification of cryptographic protocols. As a result, some tools like ProVerif (Blanchet, 2001), specially designed for this purpose rapidly overtook Timbuk. Proverif is also using approximations on cryptographic protocol modeled as sets of Horn clauses. Approximations are very efficient but statically defined in the tool itself (Blanchet, 2001). The only remaining interest in using Timbuk for cryptographic protocol verification is the fact that correctness of fixpoints (and approximations) can be certified using the checker presented in Section 4.5. This is not straightforward since the checker proposed in (Boyer et al., 2008) is defined for left-linear TRS and encoding of cryptographic protocols are not. But the result of (Boyer et al., 2008) is very likely to be extended to non left-linear TRSs.

5.2 Prototyping of static analyzers

The aim of this section is to show how to combine rewriting theory with principles from abstract interpretation in order to obtain a fast and reliable methodology for implementing static analyzers for programs. In Section 5.1 we showed that reachability analysis based

on tree automata is a powerful technique for analyzing cryptographic protocols (Genet and Klay, 2000; Feuillade et al., 2004; Genet et al., 2003). In this section, we set up a framework that allows to apply those techniques to a general programming language. Our framework consists of three parts. First, we define an encoding of the operational semantics of the language as a term rewriting system. Second, we give a translation scheme for transforming programs into rewrite rules. Finally, an over-approximation of the set of reachable program states represented by a tree automaton is computed using the tree automata completion algorithm. This framework is instantiated on a real test case, namely Java. We encode the Java Virtual Machine (JVM for short) operational semantics and Java bytecode programs into TRS and construct over-approximations of JVM states.

With regards to classical static analysis, the objective is here to use tree automata completion as a foundational mechanism for ensuring, by construction, safety of static analyzers. In particular, we aim at defining a class analysis on Java bytecode programs. Moreover, using approximation rules instead of abstract domains makes the analysis easier to fine-tune and to prove correct. This is of great interest, when a standard analysis is too coarse, since our technique permits to adapt the analysis to a particular property to prove and preserve safety. Last but not least, all the analysis performed using these technique can be certified a *unique* checker defined in Section 4.5. Actually, this is the core of the RAVAJ certified analysis chain (see Figure 5.3 and Figure 5.4).

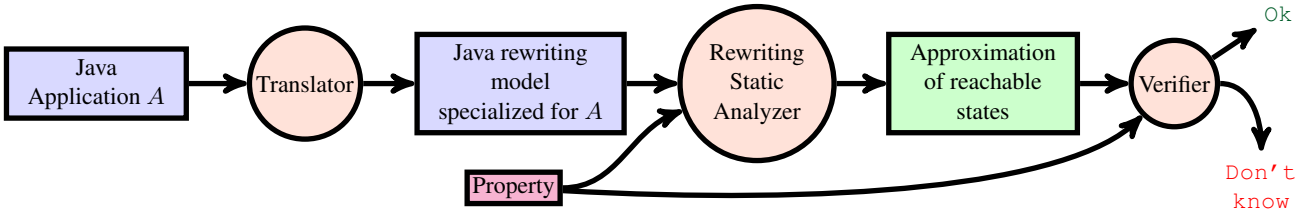


Figure 5.3: The RAVAJ Java bytecode analysis chain

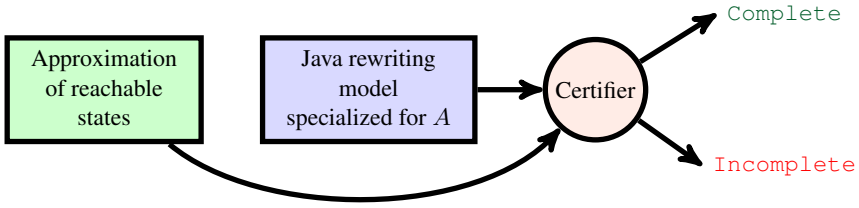


Figure 5.4: The RAVAJ Approximation checker

In the first section, we introduce the translation of Java bytecode into TRSs. Then, in the next sections, we define a class analysis and how to refine it using normalization rules. Finally, we present the Copster tool (Barré et al., 2009) that implements the translation part.

5.2.1 Translation of Java Bytecode Semantics into TRS

This section describes how to formalize the semantics of an object-oriented language (here, Java bytecode) using rewriting rules. From a bytecode Java program p, we have developed a tool named Copster (see Section 5.2.4) that automatically produces a TRS $\mathcal{R}$ modeling a significant part of the Java semantics (threads, stacks, frames, objects, references, methods, heaps, integers) as well as the semantics of p. For the moment, exceptions are not taken into account but they can be elegantly encoded using rewriting (Meseguer and Rosu, 2004; Farzan et al., 2004). The formalization follows the structure of standard Java semantics formalizations (Bertelsen, 1997; Freund and Mitchell, 1999).

Formalization of Java Program States

We first formalize semantics of Java programs without threads. A Java program state contains a current execution frame, a frame stack, a heap, and a static heap. A frame gives information about the method currently being executed: its name, current program counter, operand stack and local variables. When a method is invoked the current frame is stored in the frame stack and a new current frame is created. A heap is used to store objects and arrays, i.e. all the information that is not local to the execution of a method. The static heap stores values of static fields, i.e. values that are shared by all objects of a same class.

Let P be the infinite set of all the possible Java programs. Given $p \in P$, let $C(p)$ be the corresponding finite set of class identifiers and $C_r(p)$ be $C(p) \cup \{array\}$. A value is either a primitive type or a reference pointing to an object (or an array) in the heap. In our setting, we only consider integers, chars, strings and boolean primitive types. Let $PC(p)$ be the set of integers from 0 to the highest possible program point in all the methods in p. Let $M(p)$ be the set of method names and $M_{id}(p)$ be the finite set of pairs (m, c) where $m \in M(p)$, $c \in C(p)$ and m is a method defined by the class c. This last set is needed to distinguish between methods having the same name but defined by different classes. For the sake of simplicity, we do not distinguish between methods having the same name but a different signature but this could easily be done.

Following standard Java semantics we define a *frame* to be a tuple $f = (pc, m, s, l)$ where $pc \in PC(p)$, $m \in M_{id}(p)$, s an operand stack, l a finite map from indexes to values (local variables). An object from a class c is a map from field identifiers to values. The heap is a map from references to objects and arrays. The static heap is a map from static field names to values. A program state is a tuple $s = (f, fs, h, k)$ where f is a frame, fs is a stack of frames, h is a heap and k a static heap.

A Program State as a Term

Let $\mathcal{F}_C(p) = C(p)$ and $\mathcal{F}_{C_r}(p) = C_r(p) = C(p) \cup \{array\}$ be sets of symbols. We encode a reference as a term $loc(c, a)$ where $c \in C_r(P)$ is the class of the object being referenced and a is an integer. This is coherent with Java semantics where it is always possible to know dynamically the class of an object corresponding to a reference. We use $\mathcal{F}_{integer} = \{succ : 1, pred : 1, zero : 0\}$, $\mathcal{F}_{bool} = \{true : 0, false : 0\}$. The set $\mathcal{F}_{char}$ is built with ASCII codes of characters and $\mathcal{F}_{string} = \{Cons : 2, Nil : 0\} \cup \mathcal{F}_{char}$.

(pop)	$\dfrac{(m, pc, x :: s, l)}{(m, pc+1, s, l)}$	$(store_i)$	$\dfrac{(m, pc, x :: s, l)}{(m, pc+1, s, x \rightarrow_i l)}$

Figure 5.5: Example of bytecodes operating at the frame level

More precisely, integers are encoded using Peano's notation and strings are lists of characters built using $Cons$ and Nil operators. $\mathcal{F}_{reference}(p) = \{loc : 2, succ : 1, zero : 0\} \cup \mathcal{F}_{C_r}(p)$ for references and $\mathcal{F}_{value}(p) = \mathcal{F}_{integer} \cup \mathcal{F}_{bool} \cup \mathcal{F}_{char} \cup \mathcal{F}_{string} \cup \mathcal{F}_{reference}(p)$ for values. For example, $loc(foo, succ(zero))$ is a reference pointing to the object located at the index 1 in the foo class heap. Let x be the higher program point of the program (p), then $\mathcal{F}_{PC}(p) = \{pp0 : 0, pp1 : 0, ..., ppx : 0\}$. $\mathcal{F}_M(p)$ is defined the same way as $\mathcal{F}_C(p)$. $\mathcal{F}_{M_{id}}(p) = \{name : 2\} \cup \mathcal{F}_M(p) \cup \mathcal{F}_C(p)$. For example $name(bar, A)$ stands for the method bar defined by the class A. Let $l(p)$ denote the maximum of local variables used by the methods of the program package p. We use $\mathcal{F}_{stack}(p) = \{stack : 2, nilstack : 0\} \cup \mathcal{F}_{value}(p)$ for stacks, $\mathcal{F}_{localVars}(p) = \{locals : l(p), nillocal : 0\} \cup \mathcal{F}_{value}(p)$ for local variables and $\mathcal{F}_{frame}(p) = \{frame : 4\} \cup \mathcal{F}_{PC}(p) \cup \mathcal{F}_{M_{id}}(p) \cup \mathcal{F}_{stack}(p) \cup \mathcal{F}_{localVars}(p)$ for frames. A possible frame thus would be: $frame(name(bar, A), pp4, stack(succ(zero), nilstack), locals(loc(bar, zero), nillocal))$ where the program counter points to the 4th instruction of the method bar defined by the class A. The current operand stack has the integer 1 on the top. The first local variable is a reference and the other is not initialized.

The alphabet $\mathcal{F}_{objects}(p)$ contains the same symbols as $\mathcal{F}_C(p)$, where the arity of each symbol is the corresponding number of non-static fields. As an example, $objectA(zero)$ is an object from the class A with one field whose value is $zero$.

Let nc be the number of classes. We chose to divide the heap into nc *class heaps* plus one for the arrays. In a reference $loc(c, a)$, a is the index of the object in the list representing the heap of class c. An array is encoded using a list and indexes in a similar way. We use $\mathcal{F}_{heap}(p) = \{heaps : (nc+1), heap : 2\} \cup \mathcal{F}_{stack}(p) \cup \mathcal{F}_{objects}(p)$ for heaps, and $\mathcal{F}_{state}(p) = \{state : 4\} \cup \mathcal{F}_{frame}(p) \cup \mathcal{F}_{heap}(p)$ for states.

Java Bytecode Semantics

Figure 5.5 presents some rules of the semantics operating at the frame level. For a given instruction, if a frame matches the top expression then it is transformed into the lower expression. Considering the frame (pc, m, s, l), pc denotes the current program point, m the current method identifier, s the current stack and l the current array of local variables. The operator '::' models stack concatenation. The $store_i$ instruction is used to store the value at the top of the current stack in the ith register, where $x \rightarrow_i l$ denotes the new resulting array of local variables.

Figure 5.6 presents a rule of the semantics operating at the state level. For a state $((m, pc, s, l), fs, h, k)$, the symbols pc, m, s and l denote the current frame components, fs the current stack of frames, h the heap and k the static heap. The instruction $invokeVirtual_{name}$ implements dynamic method invocation. The method to be invoked is determined from its $name$ and the class of the reference at the top of the stack. The internal function

$$\frac{(getField_{name})\quad ((m, pc, ref :: s, l), fs, h, k), f = getf(name, ref, h, k)}{(m, pc + 1, f :: s, l)}$$

$$\frac{(invokeVirtual_{name})\quad ((m, pc, ref :: s, l), fs, h, k), c = class(ref, h, k), m' = lookup(name, c)}{((m', 0, [], storeparams(ref :: s, m')), (m, pc + 1, popparams(ref :: s, m'), l) :: fs, h, k)}$$

Figure 5.6: Example of bytecodes operating at the state level

$class(ref, h, k)$ is used to get the reference's class c and $lookup(name, c)$ searches the class hierarchy in a bottom-up fashion for the the method m' corresponding to this name and this class. There are internal functions to manage the parameters of the method (pushed on the stack before invoking): $storeparams(ref :: s, m')$ to build an array of local variables from values on the top of the operand stack and $popparams(ref :: s, m')$ to remove from the current operand stack the parameters used by m'. With those tools, it is possible to build a new frame pointing at the first program point of m' and to push the current frame on the frame stack. Some other examples can be found in (Boichut et al., 2006).

Java Bytecode Semantics using Rewriting Rules

In this section, we encode the operational semantics into rewriting rules in a way that makes the resulting system amenable to approximation. The first constraint is that the term rewriting system has to be left-linear (see Theorem 109). The second constraint, is that intermediate steps modeling internal operations of the JVM (such as low level rewriting for evaluating arithmetic operations $+, *, \ldots$), should be easy to filter out. To this end, we introduce a notion of intermediate frames (named $xframe$) encompassing all the internal computations performed by the JVM, which are not part of operational semantic rules. We can express the Java Bytecode Semantics of a Java bytecode program p by means of rewriting rules. For instance, we give here the encodings of pop, $invokeVirtual$ and $getField$ instructions.

In the following, symbols m, c, pc, s, l, fs, h, k, x, y, a, b, adr, $l0$, $l1$, $l2$, $size$, h, $h0$, $h1$, ha are variables. For a given program point pc in a given method m, we build an $xframe$ term very similar to the original $frame$ term but with the current instruction explicitly stated. The $xframe$s are used to compute intermediate steps. If an instruction requires several internal rewriting steps, we will only rewrite the corresponding $xframe$ term until the execution of the instruction ends. Assume that, in program p, the instruction at program point $pp2$ of method foo of class A is pop. In figure 5.7, Rule 1 builds an $xframe$ term by explicitly adding the current instruction to the $frame$ term. Rule 2 describes the semantics of pop. Rule 3 specifies the control flow by defining the next program point.

Now, assume that, in program p, the instruction at program point $pp2$ of method foo of class A is $invokeVirtual$. This instruction requires to compile some information about methods and the class hierarchy into the rules. Basically, we need to know what is the precise method to invoke, given a class identifier and a method name. In p, assume that A

1	$frame(name(foo, A), pp2, s, l)$	$\rightarrow$	$xframe(pop, name(foo, A), pp2, s, l)$
2	$xframe(pop, m, pc, stack(x, s), l)$	$\rightarrow$	$frame(m, next(pc), s, l)$
3	$next(pp2)$	$\rightarrow$	$pp3$

Figure 5.7: An example pop instruction by rewriting rules, for program p

and B are two classes such that B extends A. Let set be a method implemented in the class A (and thus available from B) with one parameter and $reset$ a method implemented in the class B (and thus unavailable from A) with no parameter. Figure 5.8 presents the resulting rules for this simple example.

1	$frame(name(foo, A), pp2, s, l)$	$\rightarrow$	$xframe(invokeVirtual(set),$ $name(foo, A), pp2, s, l)$
2	$state(xframe(invokeVirtual(set),$ $m, pc, stack(loc(A, adr),$ $stack(x, s)), l), fs, h, k)$	$\rightarrow$	$state(frame(name(set, A), pp0, s,$ $locals(loc(A, adr), x, nillocal)),$ $callstack(storedframe(m, pc, s, l), fs), h, k)$
3	$state(xframe(invokeVirtual(set),$ $m, pc, stack(loc(B, adr),$ $stack(x, s)), l), fs, h, k)$	$\rightarrow$	$state(frame(name(set, A), pp0, s,$ $locals(loc(B, adr), x, nillocal)),$ $callstack(storedframe(m, pc, s, l), fs), h, k)$
4	$state(xframe(invokeVirtual(reset),$ $m, pc, stack(loc(B, adr),$ $s), l), fs, h, k)$	$\rightarrow$	$state(frame(name(reset, B), pp0, s,$ $locals(loc(B, adr), nillocal, nillocal)),$ $callstack(storedframe(m, pc, s, l), fs), h, k)$
5	$next(pp2)$	$\rightarrow$	$pp3$

Figure 5.8: $invokeVirtual_{set}$ instruction by rewriting rules

The last instruction we present here, $getField$, loads the value of a field of an object stored in the heap. We use a sub rewriting system to extract the object, we are looking for, from the corresponding heap. We consider a small example of a class A with no fields and a class B with a field $field0$. In our setting, the heap is made of 3 different heaps here. The first one is for class A, the second for class B and the last one for arrays (See Figure 5.9). Rule 2 extracts from the general heap the one corresponding to class A. Rules 4 and 5 locate the object corresponding to the address. Rules 3 put the value of the field we were looking for, on the stack.

To complete the modeling of the semantics and the program by rewriting rules we need stubs for native libraries used by the program. At present, we have developed stubs for some of the methods of $java.io.InputStream$, $java.io.PrintStream$ and $java.lang.String$ classes. We model interactions of a Java program state with its environment using a term of the form $IO(s, i, o)$ where s is the state, i is the input stream and o the output stream.

Encoding threads in TRS semantics

The above TRS semantics for Java bytecode does not deal with threads. To cover threads it is necessary to adapt the encoding and add the semantics of several bytecodes. First,

1	$frame(name(foo, A), pp2, s, l)$	$\rightarrow$	$xframe(getField(f), name(foo, 1), pp2, s, l)$
2	$state(xframe(getField(field0),$ $m, pc, stack(loc(B, adr), s), l),$ $fs, heaps(h0, heap(size, h1), ha), k)$	$\rightarrow$	$state(xframe(xGetField(B, field0, h1,$ $size, adr), m, pc, s, l), fs,$ $heaps(h0, heap(size, h1), ha), k)$
3	$xframe(xGetField(B, field0$ $, stackB(objectB(x), h1),$ $succ(zero), zero), m, pc, s, l)$	$\rightarrow$	$frame(m, next(pc), stack(x, s), l)$
4	$xGetField(B, field0, h1,$ $succ(size), succ(adr))$	$\rightarrow$	$xGetField(B, field0, h1, size, adr)$
5	$xGetField(B, field0, stackB(x, h1),$ $succ(succ(size)), zero)$	$\rightarrow$	$xGetField(B, field0, h1, succ(size), zero)$
6	$next(pp2)$	$\rightarrow$	$pp3$

Figure 5.9: $getField$ instruction by rewriting rules

instead of having only one executable state we need one for every thread. As a result the general JVM executable state becomes:

$$IO(state(threadlist(\mathbf{T1}, threadlist(\mathbf{T2}, ...)), h, hs, \mathbf{lt}), i, o)$$

where $\mathbf{T1}$, $\mathbf{T2}$ are threads, h, hs and $\mathbf{lt}$ are respectively the heap, the static heap and a lock table that is detailed below. Each thread is of the form:

$$T1 = thread(\mathbf{ref}, frame(...), callstack(storedframe(...))))$$

where the construction of each thread term is similar to the construction of term states used in single threaded programs. The only difference is the $\mathbf{ref}$, the reference of the thread object itself, stored in the first field of the thread term. To adapt the single-threaded rewriting rules for dealing with multi-thread programs, it is enough to transform the set of rules operating at the state level as follows:

For rewrite rules accessing the heap For example, the $getfield$ instruction. It is necessary to replace any term or subterm of the form the $state(f, cs, h, k)$ by the term $state(threadlist(thread(\mathbf{x}, f, cs), \mathbf{y}), h, k, \mathbf{tl})$ where $\mathbf{x}, \mathbf{y}$ and $\mathbf{tl}$ are variables that will respectively match the reference of the thread, the rest of the list of thread and the lock table. For instance, the Rule 2 of the $getField$ set of rules of Figure 5.9 becomes:

2	$state(threadlist(thread(\mathbf{x}, xframe(getField(field0),$ $m, pc, stack(loc(B, adr), s), l),$ $fs), \mathbf{y}), heaps(h0, heap(size, h1), ha), k, \mathbf{tl})$	$state(threadlist(thread(\mathbf{x}, xframe(xGetField(B,$ $field0, h1, size, adr), m, pc, s, l),$ $fs), \mathbf{y}), heaps(h0, heap(size, h1), ha), k, \mathbf{tl})$

For rules accessing the call stack Those rules do not access the heap and can thus be applied independently of the other threads. For example, the $invokeVirtual$ instruction. It is enough to replace any rule rooted by $state(f, cs, h, k)$ by $thread(\mathbf{ref}, f, cs)$

where **ref** will match the reference of the thread. For instance, rule 2 of Figure 5.8 becomes:

2	$thread(\mathbf{ref}, xframe(invokeVirtual(set),$ $m, pc, stack(loc(A, adr), stack(x, s)), l), fs)$	$\rightarrow$	$thread(\mathbf{ref}, frame(name(set, A), pp0, s,$ $locals(loc(A, adr), x, nillocal)),$ $callstack(storedframe(m, pc, s, l), fs))$

Similarly, rules describing the accesses to the IO term have to be adapted so as to deal with a thread list instead of a unique thread. However, any thread which is not accessing the heap or to the in/out channels can be rewritten independently from others. This is reflected in the rules, like for $invokeVirtual$ above, that can be applied on any thread of the list. As a consequence, each thread term can be rewritten independently with rules rooted by $thread$, $frame$ and $xframe$ symbols as well as low level rules encoding the basic JVM operation, *e.g.* addition. With this encoding, a basic scheduler of threads defined as follows:

$$threadlist(x, threadlist(y, z)) \rightarrow threadlist(y, threadlist(x, z))$$

This scheduler does not guarantee liveness of the multi-threaded program but guarantee to cover every possible behavior of it. More precisely, if a thread wants to access to the heap, and thus need to be at the first position of the thread list, then there exists a rewriting derivation making it happen. In the encoding given above, the reference **ref** and the lock table are necessary for synchronization purposes. In Java, there are several ways to synchronize threads. In Copster, we only cover two of them, namely `synchronize` and `join`.

In Java, it is possible to synchronize the execution of a code block `B` by taking a lock on an object `x` using the construction `synchronized(x){B}`. When a thread T executes this instruction it tries to lock the object `x`. If T is the proprietary of the lock then it can execute `B`, otherwise it has to wait for its being free. If an object `x` is locked by a thread `T`, the lock table (represented by the term `lt` in our model) stores the fact that `x` is locked by `T` and how many times[5]. At the bytecode level, the `synchronized` instruction is replaced by two instructions `monitorenter` and `monitorexit` surrounding the bytecode translation of `B`. The JVM executes a `monitorenter` instruction for a thread `T` by checking that `T` can lock the object whose reference is on the top of the operand stack of `T`. If so, the execution can pass the *monitorenter*. Later, when the `monitorexit` is executed, the lock taken on the object is released.

The other synchronization mechanism we cover is `join`. In the code of a thread `T` it is possible to write `T2.join` which means that thread `T` waits for thread `T2` completion before continuing its execution. The encoding of the two above mechanism is rather straightforward. For instance, here is the form of the first rewrite rule encoding the join instruction.

$state(threadlist(thread(ref, frame(join, pp0,$ $stack(\mathbf{x}, s), l), cs), threadlist($ $thread(\mathbf{ref2}, xframe(returnVoid, m1, pc1, s1, ll),$ $nilcallstack), ntl)), h, k, lt)$	$\rightarrow$	$state(threadlist(thread(ref, xframe($ $xJoin(\mathbf{eqRef}(\mathbf{x}, \mathbf{ref2})), join, pp0, s, l), cs),$ $threadlist(thread(ref2, xframe(returnVoid,$ $m1, pc1, s1, ll), nilcallstack), ntl)), h, k, tc)$

[5]In Java, a thread can lock an object several times.

The intuition behind this rule is the following. If the first thread of the list is expected to $join$ another thread whose reference is x then, we look for a reference $ref2$ of a completed thread (a thread whose $xframe$ is executing the $returnVoid$ method and whose frame stack is empty $nilcallstack$). Then, the role of the $xJoin(eqRef(ref1, ref2))$ is to check if the two reference are equal. In this case, the join instruction is passed.

5.2.2 Analysis in the single threaded case

In most program analyzes, it is often necessary to know the control flow graph. For Java, as for other object-oriented languages, the control flow depends on the data flow. When a method is invoked, to know which one is executed, the class of the involved object is needed. For instance, on the Java program of Figure 5.10, `x.foo()` calls `this.bar()`. To know which version of the `bar` method is called, it is necessary to know the class of `this` and thus the class of `x` in the `x.foo()` call. The method actually invoked is determined dynamically during the program run. Class analysis aims at statically determining the class of objects stored in fields and local variables, and allows to build a more precise control flow graph valid for all possible executions. Note that in this example, exceptions around `System.in.read()` are required by the Java compiler. However, they are not taken into account by the translation.

There are different standard class analyzes, from simple and fast to precise and expensive. We consider k-CFA analysis (Shivers, 1991). In these analyzes, primitive types are abstracted by the name of their type and references are abstracted by the class of the objects they point to. In 0-CFA analysis, each method is analyzed only once, without distinguishing between the different calls (and hence the arguments passed) to this method. k-CFA analyzes different calls to the same method separately, taking into account up to k frames on the top of the frame stack.

Starting from a term rewriting system $\mathcal{R}$ modeling the semantics of a Java program, and a tree automaton $\mathcal{A}$ recognizing a set of initial Java program states, we aim at computing an automaton $\mathcal{A}^k_{\mathcal{R},\alpha}$ over-approximating $\mathcal{R}^*(\mathcal{L}(\mathcal{A}))$. From the Java source program of Figure 5.10, one can obtain the files `Test.class`, `A.class` and `B.class` whose content is around 90 lines of bytecode. The TRS $\mathcal{R}$ produced by compilation of those classes is composed of 275 rewrite rules. The number of rewrite rules is linear w.r.t. the size of the bytecode files. The analysis itself is performed using Timbuk. Successively, this section details a 0-CFA, a 1-CFA and an even more precise analysis obtained using the same TRS $\mathcal{R}$ and automaton $\mathcal{A}$, but using different sets of approximation rules. On this program, the set of reachable program states is infinite (and thus approximations are necessary) because the instruction `x=System.in.read()`, reading values in the input stream, is embedded in an unbounded loop. As long as the value stored in the variable `x` is different from `0`, the execution continues. Moreover, since we want to analyze this program for any possible stream of integers, in the automaton $\mathcal{A}$ the input stream is unbounded.

0-CFA Analysis

For a 0-CFA analysis, all integers are abstracted by their type, i.e. they are defined by the following transitions in $\mathcal{A}$: $zero \to q_{int}$, $succ(q_{int}) \to q_{int}$ and $pred(q_{int}) \to q_{int}$. The input stream is also specified by $\mathcal{A}$ as an infinite stack of integers: $nilstackin \to q_{in}$ and

```
class A{
    int y;
    void foo(){this.bar();}
    void bar(){y=1;}
}
class B extends A{
    void bar(){y=2;}
}
class Test{
    public void execute(A x){
          x.foo();
    }
    public void main(String[] argv){
          A o1;
          B o2;
          int x;
      o1= new A();
      o2= new B();
      try{
        x=System.in.read();
      }
      catch (java.io.IOException e)
        { x = 0;}
     while (x != 0){
        execute(o1);
        execute(o2);
        try{
          x=System.in.read(); }
        catch (java.io.IOException e)
          { x = 0;}}
}
}
```

Figure 5.10: Java Program Example

$stackin(q_{int}, q_{in}) \rightarrow q_{in}$. Normalization rules for integers, streams and references are defined by: $[x \rightarrow y] \rightarrow [zero \rightarrow q_{int}, succ(q_{int}) \rightarrow q_{int}, pred(q_{int}) \rightarrow q_{int}, nilstackin \rightarrow q_{in}, stackin(q_{int}, q_{in}) \rightarrow q_{in}, loc(A, z) \rightarrow q_{refA}, loc(B, u) \rightarrow q_{refB}]$ where x, y, z and u are variables. The pattern $[x \rightarrow y]$ matches any new transition to normalize and the rules $loc(A, z) \rightarrow q_{refA}$ and $loc(B, u) \rightarrow q_{refB}$ merge all references to an object of the class A and an object of the class B into the states q_{refA} and q_{refB}, respectively.

The normalization rules for frames and states are built according to the principle illustrated in Figure 5.11. The frames representing two different calls to the method `m` of the class `c` are merged independently of the current state of the execution in which the method `m` is called. The set of normalization rules α is completed by giving such an approximation rule for each method of each class. Using α, we can automatically obtain a fixpoint automaton $\mathcal{A}^{145}_{\mathcal{R},\alpha}$ over-approximating the set of all reachable Java program states. The result of the 0-CFA class analysis can be obtained, for each program location (a program point in a method in a class), by asking for the possible classes for each object in the stack or in the local variables. For instance, to obtain the set of possible classes c for the object passed as parameter to the method $execute$, i.e. the possible classes for the second local variable at program point $pp0$ of $execute$, one can use the following pattern: $frame(name(execute, Test), pp0, _, locals(_, loc(\mathbf{c}, _), ...))$. The result obtained for this pattern is that there exists two possible values for **c**: q_A and q_B which are the states recognizing respectively the classes A and B. This is consistent with 0-CFA which is not able to discriminate between the two possible calls to the $execute$ method.

1-CFA Analysis

For 1-CFA, we need to refine the set of normalization rules into α'. In α' the rules on integers, the input stream and references are similar to the ones used for 0-CFA. In α', the normalization rules for states and frames are designed according to the principle illustrated in Figure 5.12. Contrary to Figure 5.11, the frames for the method m of the class c are merged if the corresponding method calls have been done from the same program point

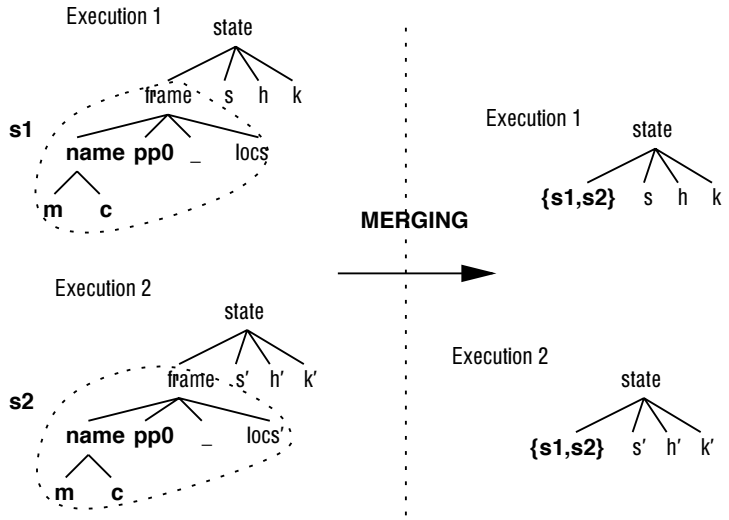


Figure 5.11: Principle of approximation rules for a 0-CFA analysis

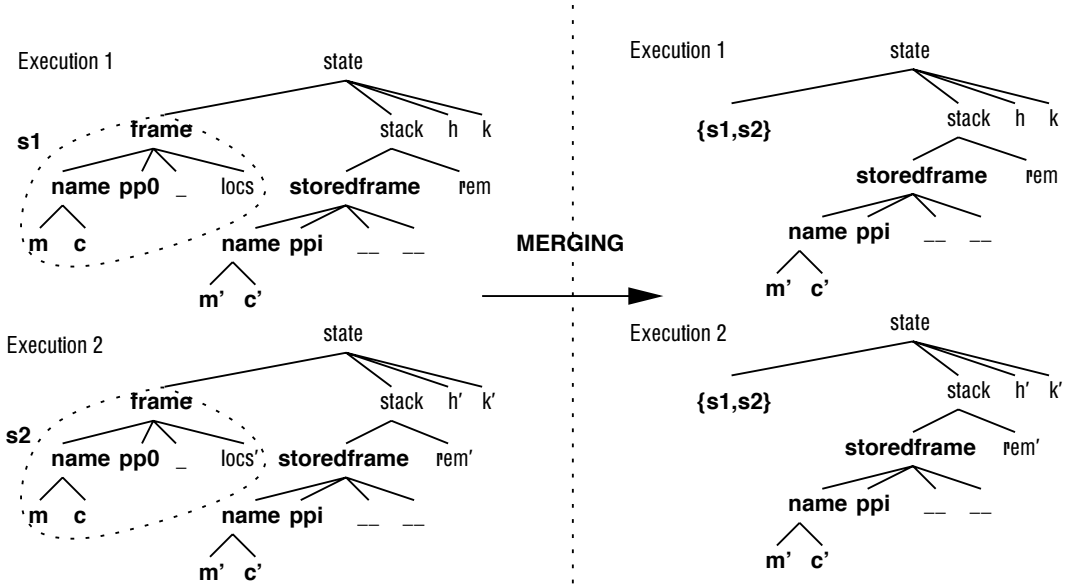


Figure 5.12: Principle of approximation rules for a 1-CFA analysis

(in the same method m' of the class c'). For example, there are two normalization rules for the method $execute$ of the class $Test$: one applying when $execute$ is invoked from the program point $pp18$ of the method $main$, and one applying when it is done from the program point $pp21$ of this same method. Applying the same principle for all the methods, we obtain a complete set of approximation rules α'. Using α', completion terminates on $\mathcal{A}^{140}_{\alpha',\mathcal{R}}$. The following patterns:

$state(frame(name(execute, Test), pp0, _, locals(_, loc(\mathbf{c}, _), ...)), stack(storeframe(_, pp18, ...), _)...)$
$state(frame(name(execute, Test), pp0, _, locals(_, loc(\mathbf{c}, _), ...)), stack(storeframe(_, pp21, ...), _)...)$

gives the desired result: each pattern has only one solution for **c**: q_A for the first and q_B for the second. Using a similar pattern to query the 0-CFA automaton $\mathcal{A}^{145}_{\mathcal{R},\alpha}$,gives q_A and q_B as solution for **c** for both program points.

Fine-tuning the precision of the analysis

Assume that we want to show that, after the execution of the previous program, field y has always a value 1 for objects of class A and 2 for objects of class B. This cannot be done by 1-CFA nor by any k-CFA since, in those analyzes, integers are abstracted by their type. One of the advantage of our technique is its ability to easily make approximation more precise by removing some approximation rules.

The property we want to prove is related to values 1 and 2 so it is tempting to refine our approximation so as not to merge those values. However, only distinguishing these two values is not enough for the analysis to succeed. Further experimentation with the approximation shows that refining the approximation of the integers by distinguishing between 0, 1, 2 and "any other integer" is enough to prove the desired property. Formally, this is expressed by the following transitions: $0 \to q_0, succ(q_0) \to q_1, succ(q_1) \to q_2, succ(q_2) \to q_{int}, succ(q_{int}) \to q_{int}$. For specifying the negative integers, the following transitions are used: $pred(q_0) \to q_{negint}$ and $pred(q_{negint}) \to q_{negint}$. The input stream representation is also modified by the following transitions: $nilstackin \to q_{in}$, $stackin(q_{negint}, q_{in}) \to q_{in}$, $stackin(q_{int}, q_{in}) \to q_{in}$ and $stackin(q_j, q_{in}) \to q_{in}$ with $j = 0, \ldots, 2$.

No other approximation is needed to ensure termination of the completion. In the fixpoint automaton $\mathcal{A}^{161}_{\mathcal{R},\alpha}$, we are then able to show that, when the Java program terminates, there are only two possible configurations of the heap. Either the heap contains an object of class A and an object of class B whose fields are both initialized to 0, or it contains an object of class A whose field has the value 1 and an object of class B whose field has the value 2. These verifications have been performed using a pattern matching with all the frames whose pp value is the last control point of the program.

This result is not surprising. The first result is possible when there is zero iterations of the loop (`x` is set to `0` before the instruction `while (x != 0){...}`). The second result is obtained for 1 or more iterations. Nevertheless, this kind of result is impossible to obtain with the two previous analyzes presented in Section 5.2.2 and 5.2.2.

5.2.3 Analysis of multi-threaded programs

Analysis of multi-threaded programs is still ongoing work. However to give a taste of what will be possible to do, we present a small multi-threaded Java program example with a

simple analysis. The program is the following.

```
class T1 extends java.lang.Thread{
    public void run(){
        while(true){
            synchronized(Top.lock){
                System.out.println(Top.f);
                int j= Top.f;
                Top.set(1);
                System.out.println(j);
                Top.set(j);
            }
        }
    }
}

class Top{
    public static Object lock;
    public static int f;
    public static void set(int i){
        f=i;
    }
    public static void main(String[] argv){
        lock = new Object();
        Top.f=0;
        for(int i=0;i<=1;i++){
            T1 t1 = new T1();
            t1.start();
        }
    }
}
```

From the bytecode of this Java program, Copster produces a TRS of 867 rewrite rules. The objective of the analysis is to prove that whatever the scheduling of threads, nothing else than 0 will be printed on the output stream. In other words, we aim at proving that the critical portion of the code which affects the `Top.f` shared variable is *really* protected by the Java `synchronize` instructions. Initially `Top.f` is 0. Each thread prints `Top.f`, set `Top.f` to 1, then sets `Top.f` to its initial value and repeat this from the beginning. If synchronization fails then several threads may execute this critical code at the same time and we are likely to print 1 values. Since threads loop for ever, an approximation equation is necessary. On this simple example, the only term that may grow infinitely is the term representing the output stream. These terms are of the form $outstack(x, outstack(y, \ldots))$ representing an output stream whose last printed element is x and y was printed immediately before. Hence, a very simple approximation equation of the form $outstack(x, outstack(y, z)) = outstack(x, z)$ is enough for the completion to terminate in less than a minute after 267 steps on a tree automaton $\mathcal{A}^{267}_{\mathcal{R},\alpha}$ having 2406 states and 9643 transitions. Then, on $\mathcal{A}^{267}_{\mathcal{R},\alpha}$ we can easily check that the pattern $outstack(succ(zero), _)$ is not found, meaning that no 1 are printed in the output stream. Finally, it is also possible to certify $\mathcal{A}^{267}_{\mathcal{R},\alpha}$ that this approximation is safe using the checker.

5.2.4 Copster

Copster (Barré et al., 2009) is a translator from Java byte code to TRS which implements the transformations defined in Section 5.2.1. Copster parses a `.class` file and produces a TRS modeling the execution of the corresponding program on the JVM. This tool was used to achieve the experiments done in the previous section. Copster is able to translate any bytecode program dealing with:

- types integers, boolean, chars, strings, objects
- class definition, inheritance, public/private/protected visibility modifiers for method definition
- basic instructions on integers, booleans, char: addition, multiplication, comparisons, ...
- basic thread mechanisms: thread creation, synchronize and join
- the following library classes and methods

```
public class java.io.IOException extends java.lang.Object{
}

public class java.io.InputStream extends java.lang.Object{
    public int read{};
}

public class java.io.PrintStream extends java.lang.Object{
    public void println{int};
    public void println{char};
    public void println{java.lang.String};
}

public class java.lang.System extends java.lang.Object{
    public static java.io.InputStream in;
    public static java.io.PrintStream out;
}

public class java.lang.String extends java.lang.Object{
    public native char charAt{int};
    public native java.lang.String concat{java.lang.String};
    public native int length{};
    public native java.lang.String substring{int};
}

public class java.lang.StringBuilder extends java.lang.Object{
}
```

5.3 Proof of strong non termination

The two previous sections describe the main applications of tree automata completion to verification namely: cryptographic protocols and Java bytecode. Those two applications are both based on reachability analysis. Before concluding this chapter, let us mention another application to verification. Tree automata completion can be used to prove strong non termination of term rewriting systems which is related to deadlock freeness of programs.

Definition 136 (Strong non-termination) *Let E be a set of terms and $\mathcal{R}$ be a TRS. The TRS $\mathcal{R}$ is said to be* strongly non-terminating *on E if there exists no finite $\mathcal{R}$-rewrite chains from terms of E.* ◇

As far as we know, this property of TRS has never been defined elsewhere in the literature. However, it is closely related to deadlock freeness property of transition systems. We now give a way of proving it on a given initial language E and a given TRS $\mathcal{R}$.

Theorem 137 *A TRS $\mathcal{R}$ is strongly non-terminating on E if $\mathcal{R}^!(E) = \emptyset$.*

PROOF. Obvious, since $\mathcal{R}^!(E) = \emptyset$ means that every term of E is reducible, and so are every terms $\mathcal{R}$-reachable from E. □

When the TRS represents some parallel processes, the strong non-termination property is close to deadlock-freeness. Let us show a very simple example of this aspect.

Example 138 *Assume that we have two processes each one having a list of elements to count. Assume that the counter is a shared variable that should not be accessed by the two processes at the same time. Each process has two possible states 'busy' if it is accessing the shared counter or 'free' otherwise. A similar flag is associated to the shared counter in order to protect it from a concurrent access. The behavior of this system is described by the following TRS $\mathcal{R}$ where x, y, z, u are variables, $Proc$ represents a process, $cons$ and $null$ are used to build the lists and S represents a configuration of the system:*

$$
\begin{array}{l}
S(Proc(free, cons(x,y)), z, free, u) \to S(Proc(busy, cons(x,y)), z, busy, u) \\
S(Proc(busy, cons(x,y)), z, busy, u) \to S(Proc(free, y), z, free, s(u)) \\
S(z, Proc(free, cons(x,y)), free, u) \to S(z, Proc(busy, cons(x,y)), busy, u) \\
S(z, Proc(busy, cons(x,y)), busy, u) \to S(z, Proc(free, y), free, s(u)) \\
S(Proc(x, null), Proc(y, null), z, u) \to S(Proc(x, null), Proc(y, null), z, u)
\end{array}
$$

The initial language E is recognized by the following tree automaton $\mathcal{A}$ whose final state is q_0 and set of transitions is:

$$
\begin{array}{lll}
S(q_1, q_1, q_2, q_3) \to q_0 & free \to q_2 & 0 \to q_3 \\
Proc(q_2, q_4) \to q_1 & null \to q_4 & cons(q_3, q_4) \to q_4
\end{array}
$$

E contains terms of the form $S(Proc(free, l_1), Proc(free, l_2), free, 0)$ where l_1 and l_2 are any lists of 0 (possibly infinite). Using the exact normalization strategy, the completion does not terminate. On the 7-th completion step the completed tree automaton $\mathcal{A}^7_{\mathcal{R},\alpha}$

contains 41 *transitions. In order to make the completion terminate, we can add an approximation equation* $s(x) = x$ *which merge some states and transition of* $\mathcal{A}^7_{\mathcal{R},\alpha}$ *together so that the tree automaton contains only* 19 *transitions. Finally* $\mathcal{A}^8_{\mathcal{R},\alpha} = \mathcal{A}^7_{\mathcal{R},\alpha}$*, hence* $\mathcal{A}^*_{\mathcal{R},\alpha} = \mathcal{A}^7_{\mathcal{R},\alpha}$*. The automaton* $\mathcal{A}^*_{\mathcal{R},\alpha}$ *over approximating* $\mathcal{R}^*(E)$ *is the following:*

```
Ops S:4 Proc:2 cons:2 null:0 busy:0 free:0 s:1 o:0
Automaton current
States qnew10:0 qnew9:0 qnew8:0 qnew7:0 qnew6:0 qnew5:0 qnew4:0
       qnew3:0 qnew2:0 qnew1:0 qnew0:0 q0:0 q1:0 q2:0 q3:0 q4:0
Final States q0
Prior
  s(qnew10) -> qnew10           Proc(qnew5,q4) -> qnew4         free -> qnew5
  Proc(qnew1,qnew3) -> qnew0    cons(qnew10,q4) -> qnew3        busy -> qnew1

Transitions
  S(q1,q1,q2,qnew10) -> q0         free -> q2
  o -> qnew10                      Proc(q2,q4) -> q1
  null -> q4                       cons(qnew10,q4) -> q4
  busy -> qnew1                    cons(qnew10,q4) -> qnew3
  Proc(qnew1,qnew3) -> qnew0       free -> qnew5
  Proc(qnew5,q4) -> qnew4          S(qnew0,qnew4,qnew1,qnew10) -> q0
  S(qnew0,q1,qnew1,qnew10) -> q0   S(qnew4,qnew0,qnew1,qnew10) -> q0
  S(q1,qnew0,qnew1,qnew10) -> q0   S(qnew4,q1,qnew5,qnew10) -> q0
  S(q1,qnew4,qnew5,qnew10) -> q0   S(qnew4,qnew4,qnew5,qnew10) -> q0
  s(qnew10) -> qnew10
```

Now, if we compute the intersection with $IRR(\mathcal{R})$ *we obtain an automaton over-approximating* $\mathcal{R}^!(E)$*. The automaton obtained by intersection recognizes an empty language. Hence, we also have* $\mathcal{R}^!(E) = \emptyset$ *and thus* $\mathcal{R}$ *is strongly non-terminating on* E*. We can thus conclude that the above system of processes is deadlock-free.*

Chapter 6

Conclusion

This document sums up our contributions in the field of software verification based on reachability analysis of term rewriting systems. This technique is rooted in the tree automata completion technique whose purpose is to compute exactly, when possible, or to over-approximate the set of reachable terms. We are conscious that this technique is not mature enough to be considered as an alternative to any well-established verification technique like model-checking or static analysis. Nevertheless, the aim of the work and experiments reported here is to demonstrate that tree automata completion has clearly some strong advantages for this purpose.

TRS as program models This strong point is shared with other verification techniques using TRS as formal models. TRS revealed to be agile enough to cover many verification problems: processes, high-order functions... In particular, it covers our two main case studies (security protocols and Java bytecode) which are of very different nature w.r.t to their abstraction level, computational model and verification objective.

Reachability analysis is exact when possible A *unique* algorithm, completion with the exact normalization strategy, covers *most* of the known classes for which an exact construction is possible. Those classes are covered by the standard and the equational tree automata completion algorithm. Furthermore, the implementations of those algorithms benefits from the optimizations performed on completion in general.

Approximations are correct by construction Each time that completion terminates, the resulting tree automaton is an over-approximation of reachable terms *whatever* the normalization rules or approximation equations may be. This is not always the case in general abstract interpretation nor in equational based abstractions where additional hand proofs are needed. In addition, all approximations computed by completion tools can be verified using a *common* certified checker (as shown in Section 4.5).

Approximations are declarative and can be tuned The great difference with usual static analysis techniques is that approximations are not built-in but, on the opposite, are defined using a declarative language (normalization rules or equations). Once defined and if necessary, these approximations can be refined by hand and adapted to a specific program or property to prove.

A step towards precision of the approximations with equations Sets of reachable terms are computed as precisely as possible on decidable classes. Normalization rules are very efficient and precise way to define approximation but, their precision is hard to estimate. When using approximation equations, no more than terms reachable by $\mathcal{R}$ modulo E are added to the completed automaton.

Completion can be scaled up to real size problems With reasonable optimizations, we managed to improve the efficiency of the Timbuk prototype so as to deal with an industrial cryptographic protocol and a detailed encoding of the JVM. Furthermore, encodings of the problem into rewriting tools (Balland et al., 2008) and logic programs static analyzers (Gallagher and Rosendahl, 2008) shows that efficiency can still be improved a lot.

On the other hand, as it is presented in this document, there is still room for further improvements to this technique. In the following, we will see that most of these points have already been addressed in other works or are ongoing research.

Discriminating between counterexamples and false positive When the approximation is successfully built but the intersection with bad configuration is not empty, two situations are possible: we have a counterexample or this term is in the over-approximation but is not reachable (false positive). It should be possible to easily discriminate between the two so as to correct the specification or correct the approximation.

Tuning approximation require expertise We eased the definition of approximations using equations rather than normalization rules. We tackled the initial objective that was to let the user define an approximation without knowing about the tree automata structure. Definition of approximations by hand using equations is more intuitive, however, it is still a trial and error process. When completion diverges this means that normalization rules or equations are incomplete and that there exists an infinite rewriting or a sequence of rewrite rules that are recursively applied on an infinite tree. However, with the current Timbuk prototype, such positions are difficult to locate and, thus, tuning of approximation require expertise.

Ensuring termination of completion before running it Approximation computed by completion are safe if completion terminates. However, for the moment, there is no way to predict its termination. Again, building a set of normalization rules or equations ensuring termination is a trial and error process.

Reachability is not enough The technique proposed here only covers reachability properties. However, temporal properties (like liveness) are of particular interest in verification.

For discriminating between counterexamples and false positive, it is possible to use critical pairs computed during completion (Boichut and Genet, 2006). These first results show that, with the epsilon-free completion algorithm of Section 3.1 and from a term t found in the approximation, if t is reachable then it is possible to build the rewriting derivation backward up to an initial term. If t is not reachable this process may diverge. We think that

the process of (Boichut and Genet, 2006) can be improved and considerably simplified by using the new completion algorithm of Section 3.2.3 where an abstraction of the rewriting path is stored in the graph of epsilon transitions.

For the tuning of approximations, there are several research directions. First, an automatic refinement is possible. This is investigated in (Boichut et al., 2008) for normalization rules. We believe that this approach can be extended to equations as follows. If a term t (a bad configuration) is found in the approximation and is not reachable, we can reconstruct backward the rewriting graph until we find that an approximation equation has been applied. Then the corresponding equation is constrained by a disequality preventing that it can be applied in that case. Then, completion is run again. However, automatic refinement is unlikely to succeed in all the cases and we need tools to ease the manual tuning of approximations. As discussed above, the replacement of normalization rules by equations is a step in that direction. What is still missing in the Timbuk prototype is an efficient tool to locate divergence sites. Some attempts of tree automata visualization and graphical merging on terms were carried out in the Tabi tree automata browsing interface (see Section 4.1.4). This solution behave well on small examples but are clearly not sufficient for real size verification problems. A research direction is thus to define a divergence visualization tool automatically focusing on divergence sites and providing enough information to guide approximation equation definition.

Termination of completion can be formally proven for a specific approximation strategy, like for the *ancestor* approximation (Genet, 1998). With normalization rules, termination of completion can be enforced using drastic rules placed at the end of the approximation specification and acting as default rules. Rules like `[x -> y] -> [z -> q]` normalizing any transition subterm by the same state `q` guarantee completion termination. However, given a set of normalization rules, proving that it ensures completion termination is hard. The situation is better with approximation equations because their semantics is formal. Recall that equations define equivalence classes of terms. A sufficient condition for completion to terminate is that the set of equivalence classes defined by E is finite. Given a set of equations E, this problem is not decidable in general (Tison, 2008) and is a collateral result of (Filiot and Tison, 2008). However, as remarked by (Tison, 2008), this problem becomes decidable if the set of equations E can be oriented into a left-linear terminating and confluent TRS $\mathcal{R}$. In that case, since $\mathcal{R}$ can be used to decide $=_E$, the set $IRR(\mathcal{R})$ has as many normal form as equivalence classes in $\mathcal{T}(\mathcal{F})/_{=_E}$. If $\mathcal{R}$ is left linear then it is possible to build a tree automaton recognizing $IRR(\mathcal{R})$ and check that this tree automaton recognize a finite language. Though the problem is undecidable in general, an interesting research direction is to define a criterion on E sufficient for $\mathcal{T}(\mathcal{F})/_{=_E}$ to be finite.

Unreachability proofs, corresponding to safety properties, are central but not always enough. Proving temporal properties (LTL properties for instance) on the rewriting graph, like it is done in (Meseguer et al., 2003), is also of great interest. Using the epsilon free completion algorithm of Section 3.1, tackling this kind of properties is uneasy because no information on the order of reachable terms is kept in the approximation automaton. Recently, Courbis et al. showed that it is possible using several approximation construction to mimic the behavior of temporal connectives. Besides to this, with the new completion algorithm of Section 3.2.3, the graph of epsilon transition represents an abstraction of the rewriting graph. For instance, on Example 138 with 8 exact steps of equational completion,

we obtain the rewriting graph of Figure 6.1 between states of the completed automaton. On this graph it is possible to prove temporal properties. Similarly, using equational approxi-

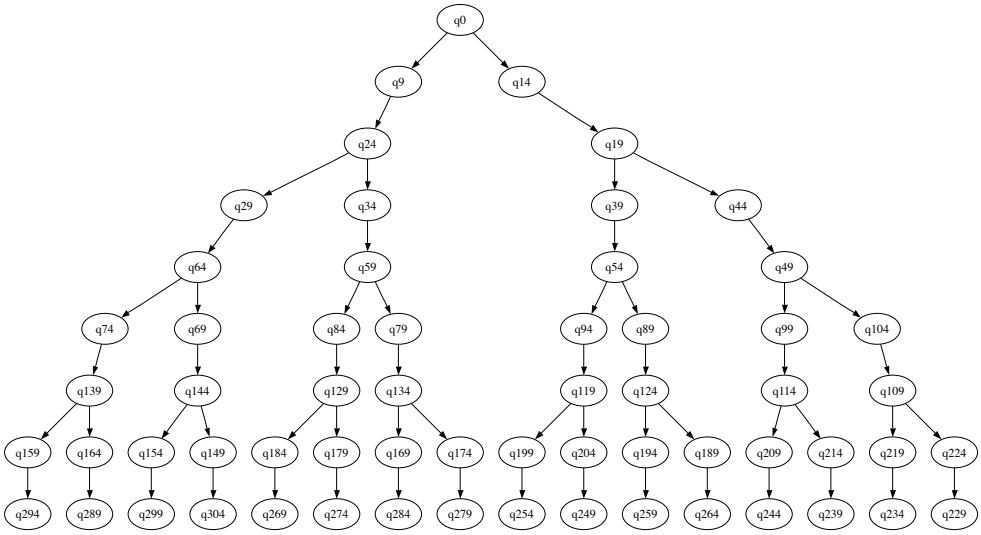


Figure 6.1: A rewriting graph between states exported by Timbuk 3.0

mation makes it possible to complete the automaton and obtain an *over-approximation* of the rewriting graph (see Figure 6.2). In (Boyer and Genet, 2009), we take advantage of this

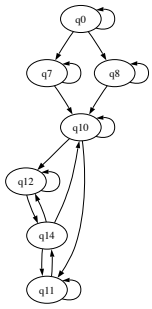


Figure 6.2: An over-approximation of a rewriting graph between states exported by Timbuk 3.0

to build from *the same* completed automaton a Kripke structure on which LTL properties can be proven. For the moment, the construction is limited to finite rewriting graphs but is likely to be extended to infinite rewriting graphs with approximations.

This last research direction is more long term. However, it should significantly widen the scope of properties covered by tree automata completion: from safety properties to liveness properties. Our aim is, of course, to benefit from tree automata over-approximations

to ease the proof of temporal properties on infinite models which is a challenging problem. Furthermore, it should be possible to combine this research with our results on Java program verification so as to prove temporal properties on multi-threaded Java programs. The static analysis of this kind of programs, using abstract interpretation, is already a very active research field. For the moment, our contribution is preliminary but shows that reachability analysis of Java multi-threaded programs using tree automata completion is possible. Like for single-threaded Java, we first aim at prototyping classical static analysis using equations. A second improvement would be to prove temporal properties on over-approximations of multi-threaded Java programs.

Bibliography

Abdulla, P. A., Legay, A., Rezine, A., d'Orso, J., 2005. Simulation-based iteration of tree transducers. In: TACAS. Vol. 3440 of LNCS. Springer, pp. 30–40.

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., Vigneron, L., 2005. The AVISPA Tool for the automated validation of internet security protocols and applications. In: CAV'2005. Vol. 3576 of LNCS. Springer, pp. 281–285.

Baader, F., Nipkow, T., 1998. Term Rewriting and All That. Cambridge University Press.

Balland, E., Boichut, Y., Genet, T., Moreau, P.-E., 2008. Towards an Efficient Implementation of Tree Automata Completion. In: AMAST'08. Vol. 5140 of LNCS.

Barré, N., Besson, F., Genet, T., Hubert, L., Le Roux, L., 2009. Copster homepage. `http://www.irisa.fr/lande/genet/copster`.

Barthe, G., Dufay, G., 2004. A tool-assisted framework for certified bytecode verification. In: FASE'04. Vol. 2984 of LNCS. Springer, pp. 99–113.

Bertelsen, P., 1997. Semantics of Java Byte Code. Tech. rep., Technical University of Denmark.
URL `citeseer.ist.psu.edu/bertelsen97semantics.html`

Bertot, Y., Castéran, P., 2004. Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions. Texts in Theoretical Computer Science. Springer Verlag.

Besson, F., Jensen, T., Pichardie, D., 2006. Proof-carrying code from certified abstract interpretation and fixpoint compression. Theor. Comput. Sci 364 (3), 273–291.

Blanchet, B., 2001. Abstracting Cryptographic Protocols by Prolog Rules. In: In 8th International Static Analysis Symposium (SAS'01). Vol. 2126 of LNCS. Springer-Verlag, pp. 433–436.

Boichut, Y., 2005. T44sp. Http://www.univ-orleans.fr/lifo/Members/Yohan.Boichut/ta4sp.html.

Boichut, Y., Courbis, R., Héam, P.-C., Kouchnarenko, O., 2008. Finer is better: Abstraction refinement for rewriting approximations. In: Voronkov, A. (Ed.), Rewriting Techniques and Applications, 19th International Conference, RTA-08. LNCS 5117. Springer, Hagenberg, Austria, pp. 48–62.

Boichut, Y., Genet, T., 2006. Feasible Trace Reconstruction for Rewriting Approximations. In: RTA. Vol. 4098 of LNCS. Springer, pp. 123–135.

Boichut, Y., Genet, T., Jensen, T., Le Roux, L., 2006. Rewriting Approximations for Fast Prototyping of Static Analyzers. Research Report RR 5997, INRIA.

Boichut, Y., Genet, T., Jensen, T., Leroux, L., 2007. Rewriting Approximations for Fast Prototyping of Static Analyzers. In: RTA. Vol. 4533 of LNCS. Springer Verlag, pp. 48–62.

Boichut, Y., Héam, P.-C., Kouchnarenko, O., 2004. Automatic Approximation for the Verification of Cryptographic Protocols. In: Proc. AVIS'2004, joint to ETAPS'04, Barcelona (Spain).

Bolignano, D., 1996. An Approach to the Formal Verification of Cryptographic Protocols. In: ACM Conference on Computer and Communications Security. pp. 106–118.

Bouajjani, A., Habermehl, P., Rogalewicz, A., Vojnar, T., 2006a. Abstract Regular Tree Model Checking. In: Infinity'05. Vol. 149(1) of ENTCS. pp. 37–48.

Bouajjani, A., Habermehl, P., Rogalewicz, A., Vojnar, T., 2006b. Abstract regular tree model checking. ENTCS 149 (1), 37–48.

Bouajjani, A., Touili, T., 2002. Extrapolating tree transformations. In: CAV. Vol. 2404 of LNCS. Springer.

Bouajjani, A., Touili, T., 2005. On computing reachability sets of process rewrite systems. In: RTA. Vol. 3467 of LNCS. pp. 484–499.

Boyer, B., Genet, T., 2009. Verifying Temporal Regular properties of Abstractions of Term Rewriting Systems. In: Proc. of RULE'09.

Boyer, B., Genet, T., Jensen, T., 2008. Certifying a Tree Automata Completion Checker. In: IJCAR'08. Vol. 5195 of LNCS. Springer.

Brainerd, W. S., 1969. Tree generating regular systems. Information and Control 14, 217–231.

Cachera, D., Jensen, T., Pichardie, P., Rusu, V., 2005. Extracting a data flow analyser in constructive logic. Theor. Comput. Sci. 342 (1), 56–78.

Catherine Meadows, 2001. Open Issues in Formal Methods for Cryptographic Protocol Analysis. LNCS 2052, 237–250.

Chevalier, Y., Vigneron, L., 2002. Automated unbounded verification of security protocols. In: CAV'02. Vol. 2404 of LNCS. Springer, pp. 324–337.

Clarke, E. M., Jha, S., Marrero, 1998. Using state space exploration and a natural deduction style message derivation engine to verify security protocols. In: In Proc. IFIP Working Conference on Programming Concepts and Methods (PROCOMET.

Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Quesada, J. F., 2001. Maude: Specification and programming in rewriting logic. Theoretical Computer Science.

Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Quesada, J. F., 2009. Maude homepage. `http://maude.cs.uiuc.edu`.

Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Talcott, C., 2008. Maude manual, version 2.4. SRI.

Comon, H., Dauchet, M., Gilleron, R., Jacquemard, F., Lugiez, D., Löding, C., Tison, S., Tommasi, M., 2008. Tree automata techniques and applications, `http://tata.gforge.inria.fr`.

Coquidé, J., Dauchet, M., Gilleron, R., Vágvölgyi, S., 1991. Bottom-up tree pushdown automata and rewrite systems. In: Book, R. V. (Ed.), Proc. 4th RTA Conf., Como (Italy). Vol. 488 of LNCS. Springer-Verlag, pp. 287–298.

Courbis, R., Héam, P.-C., Kouchnarenko, O., 2009. TAGED approximations for veriying temporal patterns. In: Proc. of CIAA'09. LNCS. Springer-Verlag.

Cousot, P., Cousot, R., 1977. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: POPL. pp. 238–252.

Cousot, P., Cousot, R., 1995. Formal Language, Grammar and Set-Constraint-Based Program Analysis by Abstract Interpretation. In: Conference Record of FPCA'95 SIGPLAN/SIGARCH/WG2.8. ACM Press, pp. 170–181.

Dauchet, M., Tison, S., Jun. 1990. The theory of ground rewrite systems is decidable. In: Proc. 5th LICS Symp., Philadelphia (Pa., USA). pp. 242–248.

D.Dolev, Yao, A., 1983. On the security of public key protocols. In: Proc. IEEE Transactions on Information Theory. pp. 198–208.

Denker, G., Meseguer, J., Talcott, C., 1998. Protocol Specification and Analysis in Maude. In: Proc. 2nd WRLA Workshop, Pont à Mousson (France).

Dershowitz, N., Jouannaud, J.-P., 1990. Handbook of Theoretical Computer Science. Vol. B. Elsevier Science Publishers B. V. (North-Holland), Ch. 6: Rewrite Systems, pp. 244–320, also as: Research report 478, LRI.

Dershowitz, N., Okada, M., Sivakumar, G., May 1988. Canonical conditional rewrite systems. In: Proc. 9th CADE Conf., Argonne (Ill., USA). Vol. 310 of LNCS. Springer-Verlag.

Durand, I., 2005. A tool for term rewrite systems and tree automata. ENTCS 124 (2), 29–49.

Durand, I., 2006. Autowrite. Http://dept-info.labri.u-bordeaux.fr/˜idurand/autowrite/.

Durand, I., Middeldorp, A., 1997. Decidable call by need computations in term rewriting. In: CADE. Vol. 1249 of LNCS. Springer, pp. 4–18.

DVB, 2003. Digital Video Broadcasting Project. http://www.dvb.org.

Escobar, S., Meseguer, J., 2007. Symbolic model checking of infinite-state systems using narrowing. In: RTA'07. Vol. 4533 of LNCS. Springer, pp. 153–168.

Farzan, A., Chen, C., Meseguer, J., Rosu, G., 2004. Formal analysis of java programs in javafan. In: CAV. Vol. 3114 of Lecture Notes in Computer Science. Springer, pp. 501–505.

Feuillade, G., Genet, T., June 2003. Reachability in conditional term rewriting systems. In: FTP'2003, International Workshop on First-Order Theorem Proving. Vol. 86 n. 1 of ENTCS. Elsevier.

Feuillade, G., Genet, T., Viet Triem Tong, V., 2004. Reachability Analysis over Term Rewriting Systems. Journal of Automated Reasonning 33 (3-4), 341–383.
URL http://www.irisa.fr/lande/genet/publications.html

Filiot, E., Tison, S., 2008. Regular n-ary queries in trees and variable independence. In: 5th IFIP International Conference on Theoretical Computer Science.

Freund, S. N., Mitchell, J. C., 1999. A formal framework for the Java bytecode language and verifier. ACM SIGPLAN Notices 34 (10), 147–166.
URL citeseer.ist.psu.edu/freund99formal.html

Gallagher, J., Rosendahl, M., 2008. Approximating term rewriting systems: a horn clause specification and its implementation. LNCS 5330.

Gécseg, F., Steinby, M., 1984. Tree automata. Akadémiai Kiadó, Budapest, Hungary.

Genet, T., 1997. Decidable approximations of sets of descendants and sets of normal forms (extended version). Tech. Rep. RR-3325, INRIA.
URL file://ftp.loria.fr/pub/loria/protheo/TECHNICAL_REPORTS_1997/Genet-RR97.ps.gz

Genet, T., 1998. Decidable approximations of sets of descendants and sets of normal forms. In: Proc. 9th RTA Conf., Tsukuba (Japan). Vol. 1379 of LNCS. Springer-Verlag, pp. 151–165.

Genet, T., 2003. Timbuk 2.0 – a Tree Automata Library – Reference Manual and Tutorial. IRISA / Université de Rennes 1, 81 pages. http://www.irisa.fr/lande/genet/timbuk/.

Genet, T., Klay, F., 2000. Rewriting for Cryptographic Protocol Verification. In: Proc. 17th CADE Conf., Pittsburgh (Pen., USA). Vol. 1831 of LNAI. Springer-Verlag.

Genet, T., Rusu, R., 2009. Equational tree automata completion. Journal of Symbolic ComputationSubmitted.

Genet, T., Tang-Talpin, Y.-M., Viet Triem Tong, V., 2003. Verification of Copy Protection Cryptographic Protocol using Approximations of Term Rewriting Systems. In: WITS'2003.

Genet, T., Viet Triem Tong, V., 2001. Reachability Analysis of Term Rewriting Systems with *timbuk*. In: Proc. 8th LPAR Conf., Havana (Cuba). Vol. 2250 of LNAI. Springer-Verlag, pp. 691–702.
URL `ftp://ftp.irisa.fr/local/lande/tg-vvtt-lpar01.ps.gz`

Genet, T., Viet Triem Tong, V., 2002. Proving Negative Conjectures on Equational Theories using Induction and Abstract Interpretation. Tech. Rep. RR-4576, INRIA.
URL `ftp://ftp.irisa.fr/local/lande/tg-vvtt-inria02.ps.gz`

Gilleron, R., Tison, S., 1995. Regular tree languages and rewrite systems. Fundamenta Informaticae 24, 157–175.

Goubault-Larrecq, J., 2000. A method for automatic cryptographic protocol verification. In: In Proceedings of 15th IPDPS Workshop. Vol. 1800 of LNCS. pp. 977–984.

Gyenizse, P., Vágvölgyi, S., 1998. Linear Generalized Semi-Monadic Rewrite Systems Effectively Preserve Recognizability. TCS 194(1-2), 87–122.

Heen, O., Genet, T., Geller, S., Prigent, N., 2008. An industrial and academical joint experiment on automated verification of a security protocol. In: IFIP MWNS'08 Workshop.

Heintze, N., 1993. Set constraints in program analysis. In: Worskop ISLP.

Heintze, N., Jaffar, J., 1990. A finite presentation theorem for approximating logic programs. In: POPL. pp. 197–209.

Jacquemard, F., 1996. Decidable approximations of term rewriting systems. In: Ganzinger, H. (Ed.), Proc. 7th RTA Conf., New Brunswick (New Jersey, USA). Springer-Verlag, pp. 362–376.

Jacquemard, F., Rusinowitch, M., Vigneron, L., 2000. Compiling and Verifying Security Protocols. In: Proceedings of 7th Conference on Logic for Programming and Automated Reasoning. Vol. 1955 of LNAI. Springer-Verlag.

Jones, N. D., 1987. Flow Analysis of Lazy Higher-Order Functional Programs. In: Abramsky, S., Hankin, C. (Eds.), Abstract Interpretation of Declarative Languages. Ellis Horwood, Chichester, England, pp. 103–122.

Jones, N. D., Andersen, N., 2007. Flow analysis of lazy higher-order functional programs. TCS 375 (1-3), 120–136.

Jones, N. D., Muchnick, S. S., 1979. Flow analysis and optimization of lisp-like structures. In: POPL. pp. 244–256.

Klein, G., Nipkow, T., 2003. Verified bytecode verifiers. TCS 298.

Leroy, X., Doligez, D., Garrigue, J., Rémy, D., Vouillon, J., 2000. The Objective Caml system release 3.00 – Documentation and user's manual. INRIA, `http://caml.inria.fr/ocaml/htmlman/`.

Letouzey, P., Théry, L., 2000. Formalizing stalmarck's algorithm in coq. In: Proc. of TPHOL'00. Vol. 1869 of LNCS. Springer.

Lowe, G., 1996a. Breaking and fixing the Needham-Schroeder public-key protocol using CSP and FDR. In: Proc. 2nd TACAS Conf., Passau (Germany). Vol. 1055 of LNCS. Springer-Verlag, pp. 147–166.

Lowe, G., 1996b. Some New Attacks upon Security Protocols. In: 9th Computer Security Foundations Workshop. IEEE Computer Society Press.

Mayr, R., 1998. Decidability and complexity of model checking problems for infinite-state systems. Ph.D. thesis, TUM.

Meseguer, J., Palomino, M., Martí-Oliet, N., 2008. Equational abstractions. TCS 403 (2-3), 239–264.

Meseguer, J., Palomino, M., Martí-Oliet, N., 2003. Equational Abstractions. In: Proc. 19th CADE Conf., Miami Beach (Fl., USA). Vol. 2741 of LNCS. Springer, pp. 2–16.

Meseguer, J., Rosu, G., 2004. Rewriting logic semantics: From language specifications to formal analysis tools. In: IJCAR. pp. 1–44.

Monniaux, D., 1999. Abstracting Cryptographic Protocols with Tree Automata. In: Proc. 6th SAS, Venezia (Italy).

Nagaya, T., Toyama, Y., 1999. Decidability for left-linear growing term rewriting systems. In: RTA. Vol. 1631 of LNCS. Springer, pp. 256–270.

Needham, R. M., Schroeder, M. D., 1978. Using Encryption for Authentication in Large Networks of Computers. CACM 21 (12), 993–999.

Nesi, M., Rucci, G., Verdesca, M., 2003. A rewriting strategy for protocol verification. In: WRS'03. Vol. 84–4 of ENTCS. pp. 65–78.

Oehl, F., Cécé, G., Kouchnarenko, O., Sinclair, D., 2003. Automatic Approximation for the Verification of Cryptographic Protocols. In: Proc. of FASE'03. Vol. 2629 of LNCS. Springer-Verlag, pp. 34–48.

Oehl, F., Sinclair, D., 2001. Combining two approaches for the formal verification of cryptographic protocols. In: Proceedings of ICLP Workshop on Specification, Analysis and Validation for Emerging technologies in computational logic.

Patin, G., Sighireanu, M., Touili, T., 2007a. Spade: Verification of multithreaded dynamic and recursive programs. In: Proc. of CAV'07. Vol. 4590 of LNCS. Springer, pp. 254–257.

Patin, G., Sighireanu, M., Touili, T., 2007b. SPADE: Verification of multithreaded dynamic and recursive programs. Http://www.liafa.jussieu.fr/˜touili/spade.html.

Paulson, L., 1997. Proving Properties of Security Protocols by Induction. In: 10th Computer Security Foundations Workshop. IEEE Computer Society Press.

Réty, P., 1999. Regular Sets of Descendants for Constructor-based Rewrite Systems. In: Proc. 6th LPAR Conf., Tbilisi (Georgia). Vol. 1705 of LNAI. Springer-Verlag.

Réty, P., Vuotto, J., 2002. Regular Sets of Descendants by some Rewrite Strategies. In: Proc. 13th RTA Conf., Copenhagen (Denmark). Vol. 2378 of LNCS. Springer-Verlag.

Reynolds, J., 1969. Automatic computation of data set definitions. Information Processing 68, 456–461.

Rusinowitch, M., Turuani, M., 2001. Protocol insecurity with finite number of sessions is np-complete. In: CSFW. IEEE Computer Society.

Salomaa, K., 1988. Deterministic Tree Pushdown Automata and Monadic Tree Rewriting Systems. J. of Computer and System Sciences 37, 367–394.

Seki, H., Takai, T., Fujinaka, Y., Kaji, Y., 2002. Layered transducing term rewriting system and its recognizability preserving property. In: Proc. 13th RTA Conf., Copenhagen (Denmark). Vol. 2378 of LNCS. Springer-Verlag.

Shivers, O., June 1991. The semantics of Scheme control-flow analysis. In: Proceedings of the Symposium on Partial Evaluation and Semantics-Based Program Manipulation. Vol. 26. New Haven, CN, pp. 190–198.

Sighireanu, M., Touili, T., 2006. PRESS: analysis of process rewrite systems. Http://www.liafa.jussieu.fr/˜sighirea/press/.

SmartRight, October 2001. Smartright Technical White Paper v1.0. Thomson, `http://www.smartright.org`.

Takai, T., 2004. A Verification Technique Using Term Rewriting Systems and Abstract Interpretation. In: Proc. 15th RTA Conf., Aachen (Germany). Vol. 3091 of LNCS. Springer, pp. 119–133.

Takai, T., Kaji, Y., Seki, H., 2000. Right-linear finite-path overlapping term rewriting systems effectively preserve recognizability. In: Proc. 11th RTA Conf., Norwich (UK). Vol. 1833 of LNCS. Springer-Verlag.

Tison, S., 2008. Personnal communication.

Tom, 2009. Tom Homepage. `http://tom.loria.fr`.

Turuani, M., 2003. Security of cryptographic protocols: Decidability and complexity. Ph.D. thesis, Université of Nancy 1.

Viry, P., 2002. Equational rules for rewriting logic. Theoretical Computer Science 285, 487–517.

Whaley, J., Unkel, C., Lam, M. S., 2004. A bdd-based deductive database for program analysis. `http://bddbddb.sourceforge.net`.

Zunino, R., Degano, P., 2006. Handling exp, $\times$ (and timestamps) in protocol analysis. In: Proc. of FOSSACS'06. Vol. 3921 of LNCS. Springer, pp. 413–427.